人工智能语言－－

Prolog 语言教程

素材来源于：中国人工智能网

闫净斌收集整理

2011－10－27

目 录

第 0 章 人工智能语言—PROLOG 简介

人工智能语言是一类适应于人工智能和知识工程领域的、具有符号处理和逻辑推理能力的计算机程序设计语言,其中 Prolog 是当代最有影响的人工智能语言之一。

一、什么是人工智能语言

人工智能（AI）语言是一类适应于人工智能和知识工程领域的、具有符号处理和逻辑推理能力的计算机程序设计语言。能够用它来编写程序求解非数值计算、知识处理、推理、规划、决策等具有智能的各种复杂问题。

典型的人工智能语言主要有 LISP、Prolog、Smaltalk、C++等。

一般来说，人工智能语言应具备如下特点：

- 具有符号处理能力（即非数值处理能力）；
- 适合于结构化程序设计，编程容易；
- 具有递归功能和回溯功能；
- 具有人机交互能力；
- 适合于推理；
- 既有把过程与说明式数据结构混合起来的能力，又有辨别数据、确定控制的模式匹配机制。

人们可能会问，用人工智能语言解决问题与传统的方法有什么区别呢？

传统方法通常把问题的全部知识以各种的模型表达在固定程序中，问题的求解完全在程序制导下按着预先安排好的步骤一步一步（逐条）执行。解决问题的思路与冯.诺依曼式计算机结构相吻合。当前大型数据库法、数学模型法、统计方法等都是严格结构化的方法。

对于人工智能技术要解决的问题，往往无法把全部知识都体现在固定的程序中。通常需要建立一个知识库（包含事实和推理规则），程序根据环境和所给的输入信息以及所要解决的问题来决定自己的行动，所以它是在环境模式的制导下的推理过程。这种方法有极大的灵活性、对话能力、有自我解释能力和学习能力。这种方法对解决一些条件和目标不大明确或不完备，（即不能很好地形式化，不好描述）的非结构化问题比传统方法好，它通常采用启发式、试探法策略来解决问题。

二、Prolog 语言及其基本结构

Prolog 是当代最有影响的人工智能语言之一，由于该语言很适合表达人的思维和推理规则，在自然语言理解、机器定理证明、专家系统等方面得到了广泛的应用，已经成为人工智能应用领域的强有力的开发语言。

尽管 Prolog 语言有许多版本，但它们的核心部分都是一样的。Prolog 的基本语句仅有三种，即事实、规则和目标三种类型的语句，且都用谓词表示，因而程序逻辑性强，文法简捷，清晰易懂。另一方面，Prolog 是陈述性语言，一旦给它提交必要的事实和规则之后，Prolog 就使用内部的演绎推理机制自动求解程序给定的目标，而不需要在程序中列出详细的求解步骤。

１、事实

事实用来说明一个问题中已知的对象和它们之间的关系。在 Prolog 程序中，事实由谓词名及用括号括起来的一个或几个对象组成。谓词和对象可由用户自己定义。

例如，谓词 likes(bill，book).

是一个名为 like 的关系，表示对象 bill 和 book 之间有喜欢的关系。

２、规则

规则由几个互相有依赖性的简单句（谓词）组成，用来描述事实之间的依赖关系。从形式上看，规则由左边表示结论的后件谓词和右边表示条件的前提谓词组成。

例如，规则 bird(X):-animal(X),has(X,feather).

表示凡是动物并且有羽毛，那么它就是鸟。

３、目标（问题）

把事实和规则写进 Prolog 程序中后，就可以向 Prolog 询问有关问题的答案，询问的问题就是程序运行的目标。目标的结构与事实或规则相同，可以是一个简单的谓词，也可以是多个谓词的组合。目标分内、外两种，内部目标写在程序中，外部目标在程序运行时由用户手工键入。

例如问题 ?-student(john).

表示“john 是学生吗？”

三、Prolog 程序的简单例子

以下两个例子在 Turbo Prolog 2.0 环境下运行通过。

[注：一个 Turbo Prolog 程序至少包括谓词段、子句段和目标段三项。目标可以包含在程序中，也可以在程序运行时给出。]

例 1 谁是 john 的朋友?

```
predicates /*谓词段，对要用的谓词名和参数进行说明*/

likes(symbol, symbol)

friend(symbol, symbol)

clauses /*子句段，存放所有的事实和规则*/

likes(bell,sports). /*前 4 行是事实*/

likes(mary,music).

likes(mary,sports).

likes(jane,smith).

friend(john,X):-likes(X,sports),likes(X,music). /*本行是规则*/
```

当上述事实与规则输入计算机后，运行该程序，用户就可以进行询问，如输入目标：

```
friend(john,X)
```

即询问 john 的朋友是谁,,这时计算机的运行结果为：

```
X=mary （mary 是 john 的朋友）

1 Solution （得到了一个结果）
```

程序运行界面如下图所示:

例 2 汉诺塔问题：

有 N 个有孔的盘子，最初这些盘子都叠放在柱 a 上（如图 1），要求将这 N 个盘子借助柱 b 从柱 a 移到柱 c（如图 2），移动时有以下限制：每次只能移动一个盘子；大盘不能放在小盘上。问如何移动？

该问题可以采用递归法思想来求解,其源程序为:

```
predicates /*谓词段*/
hanoi(integer)
move(integer,symbol,symbol,symbol)
inform(symbol,symbol).

clauses /*子句段*/
hanoi(N):-move(N,a,b,c).
move(1,A,_,C):-inform(A,C),!.
move(N,A,B,C):-N1=N-1,move(N1,A,C,B),
inform(A,C),move(N1,B,A,C).
inform(Loc1,Loc2):-nl,write("移动 1 个盘子从柱" ,Loc1,"到柱",Loc2).

goal /*目标段，问移动 3 个盘子的方法*/

hanoi(3).
```

这个例子的目标包含在程序里面，因此运行时程序将直接输出所有结果。

程序运行界面如下图所示：

四、Prolog 语言的常用版本

Prolog 语言最早是由法国马赛大学的 Colmerauer 和他的研究小组于 1972 年研制成功。早期的 Prolog 版本都是解释型的，自 1986 年美国 Borland 公司推出编译型 Prolog,即 Turbo Prolog 以后，Prolog 便很快在 PC 机上流行起来。后来又经历了 PDC PROLOG、Visual Prolog 不同版本的发展。并行的逻辑语言也于 80 年代初开始研制，其中比较著名的有 PARLOG、Concurrent PROLOG 等。

1、Turbo Prolog

由美国 Prolog 开发中心（Prolog Development Center, PDC）1986 年开发成功、Borland 公司对外发行，其 1.0，2.0，2.1 版本取名为 Turbo Prolog，主要在 IBM PC 系列计算机，MS-DOS 环境下运行。

2、PDC Prolog

1990 年后，PDC 推出新的版本，更名为 PDC Prolog 3.0，3.2，它把运行环境扩展到 OS/2 操作系统，并且向全世界发行。它的主要特点是:

- 速度快。编译及运行速度都很快，产生的代码非常紧凑。

- 用户界面友好。提供了图形化的集成开发环境。
- 提供了强有力的外部数据库系统。
- 提供了一个用 PDC Prolog 编写的 Prolog 解释起源代码。用户可以用它研究 Prolog 的内部机制，并创建自己的专用编程语言、推理机、专家系统外壳或程序接口。
- 提供了与其他语言（如 C、Pascal、Fortran 等）的接口。Prolog 和其他语言可以相互调用对方的子程序。
- 具有强大的图形功能。支持 Turbo C、Turbo Pascal 同样的功能。

3、Visual Prolog

Visual Prolog 是基于 Prolog 语言的可视化集成开发环境，是 PDC 推出的基于 Windows 环境的智能化编程工具。目前，Visual Prolog 在美国、西欧、日本、加拿大、澳大利亚等国家和地区十分流行，是国际上研究和开发智能化应用的主流工具之一。

Visual Prolog 具有模式匹配、递归、回溯、对象机制、事实数据库和谓词库等强大功能。它包含构建大型应用程序所需要的一切特性：图形开发环境、编译器、连接器和调试器，支持模块化和面向对象程序设计，支持系统级编程、文件操作、字符串处理、位级运算、算术与逻辑运算，以及与其它编程语言的接口。

Visual Prolog 包含一个全部使用 Visual Prolog 语言写成的有效的开发环境，包含对话框、菜单、工具栏等编辑功能。

Visual Prolog 与 SQL 数据库系统、C++开发系统、以及 Visual Basic、Delphi 或 Visual Age 等编程语言一样，也可以用来轻松地开发各种应用。

Visual Prolog 软件的下载地址为：http://www.visual-prolog.com 。

第 1 章-补充教程(写在正式教程的前面)

prolog-使用逻辑编程语言的教程，第一章主要包括概述和前言，什么是 Prolog，为什么要使用 Prolog 等。

如果你是一位 prolog 的新手，希望你首先阅读这篇文章，好对 prolog 的全局有个了解。在这篇文章中我会把 prolog 和其他的程序语言做比较，所以希望你已经具有了一定的编程水平。

什么是 prolog?

prolog 是 Programming in LOGic 的缩写，意思就是使用逻辑的语言编写程序。prolog 不是很高深的语言，相反，比较起其他的一些程序语言，例如 c、basic 等等语言， prolog 是更加容易理解的语言。如果你从来没有接触过计算机编程，那么恭喜你，你将很容易的进入 prolog 世界。如果你已经是其他语言的高手，你就需要完全丢弃你原来的编程思路，否则是很难掌握 prolog 的。

一个例子

逻辑思维在我们日常生活中比比皆是，prolog 正是把这种思维用文字描述出来的计算机语言。还是首先举个例子吧。

比如一群年轻人正在恋爱，每个人都有自己心中所追求的对象：

```
张学友爱王菲

张学友爱周慧敏

王菲爱谢廷峰

周慧敏爱张学友

谢廷峰爱王菲

谢廷峰爱周慧敏

刘德华爱周慧敏

......
```

我们说两个年轻人要互相都喜爱，他们就算是一对情侣，那么上面的谁和谁是情侣呢？

这应该算是一道最简单逻辑推理题目了，那么我们如何用 prolog 语言实现呢？

“张学友爱王菲”是一条已知的事实，用 prolog 语言来表达就是：

爱(张学友,王菲).

- 注意 1：这里是为了阅读方便才使用汉字的，真正的 prolog 是不允许使用除了基本字符以外字符的，也就是说，上面的句子必须写成 love(zhangxueyou,wanfei).，电脑才能够真正的理解。
- 注意 2：最末尾的“.”一定不能掉，它表示一个句子结束。
- 注意 3：上面词汇对于电脑来说并没有真正的含义，所以我们完全可以用 ai(zxy,wf). 来表达这个关系，更进一步，我们甚至可以用 xxx(a,b).来表达，只要你自己心里清楚 xxx 表示爱，a 表示张学友，b 表示王菲就可以了。
- 注意 4：张学友和王菲的顺序也没有特别的规定，你完全可以把他们换个位置：爱(王菲,张学友). 只要你心里清楚它表达的意思就行了，而以后都遵循这种被爱的人在前面的顺序，就不会出错。

其他的事实我就不写了，你可以参照上面的例子自己把已知事实翻译成 prolog 的语句。

那么情侣的概念怎么定义呢？也很简单！

情侣(某人甲,某人乙):-爱(某人甲,某人乙),爱(某人乙,某人甲).

:-在 prolog 中表示“如果”的意思，我们使用它来定义规则。上面这句话的意思就是，某人甲和某人乙是情侣的规则就是：某人甲爱某人乙，并且某人乙爱某人甲。上面用来分隔两个爱的句子的“，”表示并且的意思。

当然为了能够让电脑运行，这个句子要改为英文的：

lovers(X,Y):-love(X,Y),love(Y,X).

注意：在 prolog 中以小写字符开头的字符串代表确知的事物，比如 love 表示爱这种关系，而 zhangxueyou 表示张学友。而以大写字母开头的字符串表示未确定的事物，翻译成汉语就是某某。

完整的可运行的 prolog 程序如下：（我的拼音不好，要是什么人的名字拼写错了，请原谅：）

```
love(zhangxueyou,wanfei).
```

```
love(zhangxueyou,zouhuimin).

love(wanfei,xietinfen).

love(zouhuimin,zhangxueyou).

love(xietinfen,wanfei).

love(xietinfen,zouhuimin).

love(liudehua,zouhuimin).

lovers(X,Y):-love(X,Y),love(Y,X).
```

我们可以看出来，完整的 prolog 程序是有事实和规则组成的。事实用来储存一些数据，而规则用来储存某种可以推理出来的关系。

如果把上面的程序调入 prolog 解释器（关于 prolog 解释器，在后面有介绍）然后就可以对以上的程序进行询问。

prolog 解释器的提示符号为“?-”，你只需要在在这个提示符后面输入自己的句子就可以了。让我们来看第一个询问：

?-love(zhangxueyou,wanfei).

事实上我们的询问完全和程序中的第一条事实一样，这个询问是“是非”询问，也就是说电脑回答的答案是 yes 或者 no。上面的询问的含义是：就你所知，张学友爱王菲么？由于我们的程序中间有这样的事实，所以解释器将回答。

yes.

如果我们问：

?-love(zhangxueyou,liudehua).

解释器将回答

no.

因为它没有发现 love(zhangxueyou,liudehua).这个事实。

在询问中我们可以使用大写字母代表未知的事物，让解释器找到答案。例如：

?-love(zhangxueyou,X).

这句话询问的是：张学友都喜欢那些人。解释器将给出答案：

```
X=wanfei;

X=zouhuimin;

no.
```

注意 1：上面的两个“;”是人工输入的，当解释器找到一个答案之后，它将这个答案输出，并且等待用户的进一步输入，如果用户输入“；”，解释器将继续寻找其他的答案，如果输入的是别的符号，解释器将终止查询。

最后那个 no.是因为，系统在输出了 zouhuimin 这个答案以后，用户输入“；”，表示还想知道其他的答案，而解释器又找不到其他的答案了，于是输出 no.来终止查询。我们再看一个例子：

```
?-love(X,zouhuimin).

X=zhangxueyou;

X=xietinfen;

X=liudehua;

no.
```

在上面的询问中，我们只涉及到对事实的查询，下面我们来看规则的用法。

```
  ?- lovers(X,Y).

X = zhangxueyou

Y = zouhuimin ;



X = wanfei

Y = xietinfen ;



X = zouhuimin

Y = zhangxueyou ;
```

```
X = xietinfen

Y = wanfei ;



no
```

我们看到 lovers(X,Y).找出了系统中所有的恋人。不过每对恋人被显示了两次，这是因为 prolog 是考虑顺序的，也就是说 lovers(a,b).和 lovers(b,a).并不等价。这一点在后面的学习中，你会了解。

再看一个例子：

```
  ?- lovers(wanfei,Y).

Y = xietinfen ;

no
```

询问王菲的恋人，结果是 xietinfen。呵呵，还挺聪明的。我们看到同样是 lovers，根据其参数不同，功能也不同，这也是 prolog 的一个大特点。

最后让我们编写一个寻找情敌的规则来结束这一节内容吧。

```
rival_in_love(X,Y):-love(X,Z),not(love(Z,X)),love(Z,Y).
```

这段程序可以理解为：Y 是 X 的情敌的条件是：X 喜欢 Z(代表某个人)，而 Z 不喜欢 X，而 Y 是 Z 喜欢的人。哈哈，这不正是情敌的条件嘛。

```
?- rival_in_love(X,Y).

X = zhangxueyou

Y = xietinfen ;



X = xietinfen

Y = zhangxueyou ;
```

```
X = liudehua

Y = zhangxueyou ;



no
```

好了，你自己分析一下为什么会是这样的答案吧。

为什么要 prolog

看完上面的例子，不知道是否提起了你对 prolog 的兴趣。如果你感兴趣的话，那么让我们继续来看 prolog 能够做一些什么事情吧。

理论上来说使用 c 语言可以编制任何种类的程序，甚至连 prolog 语言都是使用 c 语言编写的。不过对于急于开发应用程序的用户，最关心的是如何最经济最有效率的开发程序，prolog 为你多提供了一个选择的余地。

prolog 很适合于开发有关人工智能方面的程序，例如：专家系统、自然语言理解、定理证明以及许多智力游戏。曾经有人预言 prolog 将成为下一代计算机的主要语言，虽然这个梦想目前还很难实现，不过世界上已经有许多 prolog 的应用实例了。你要坚信，它绝对不是那种只在实验室发挥作用的语言，之所以大多数人都不了解它，是因为它的应用范围比较特殊而已。

prolog 有许多不足之处，但是这并不影响它在逻辑推理方面的强大功能，不过最好的方法是使用某种一般语言和 prolog 结合，一般语言完成计算、界面之类的操作，而 prolog 则专心实现逻辑运算的操作。例如：你编写一个下棋程序，用 prolog 来让电脑思考如何下棋，而用 Visual Basic 来编写界面。我们将在以后介绍这方面的技术。

总之，prolog 在许多方面将极大的减少你的编程负担，所以赶快来了解一下它吧， 也许你日后遇到什么难题，可以使用 prolog 迎刃而解，到那个时候，你就知道今天的学习没有白费了。

prolog 的特点

我个人总结了 prolog 的以下几个特点，因为叫做特点，所以自然要和其他的语言进行比较。

1. prolog 程序没有特定的运行顺序，其运行顺序是由电脑决定的，而不是编程序的人

从这个意义上来说，prolog 程序不是真正意义上的程序。所谓程序就是按照一定的步骤运行的计算机指令，而 prolog 程序的运行步骤不由人来决定。它更像一种描述型的语言，用特定的方法描述一个问题，然后由电脑自动找到这个问题的答案。举个极端的例子，你只需要把某个数学题目告诉它，它就会自动的找到答案，而不像使用其他的语言一样，必须人工的编制出某种算法。

2. prolog 程序中没有 if、when、case、for 这样的控制流程语句

前面已经说了，程序的运行方式有电脑自己决定，当然就用不到这些控制流程的语句了。通常情况下，程序员不需要了解程序的运行过程，只需要注重程序的描述是否全面，不过 prolog 也提供了一些控制流程的方法，这些方法和其他语言中的方法有很大的区别，希望你在以后的学习当中能够融会贯通。

3. prolog 程序和数据高度统一

在 prolog 程序中，是很难分清楚哪些是程序，哪些是数据的。事实上，prolog 中的所有东西都有相同的形式，也就是说数据就是程序，程序就是数据。举一个其他语言的例子：如果想用 c 语言编写一个计算某个数学表达式的程序很简单(比如：a=2+54)，因为这是一段程序。但是如果想编写一个计算用户输入的表达式的值的程序就很困难了。因为用户输入的是一段数据(字符串)，如果想让 c 语言处理这个字符串，就需要很多方面的技术。则正是因为在 c 语言中，程序和数据是分开的。而在 prolog 就不存在这个问题，你甚至可以很轻松的编写处理其它 prolog 程序的程序。

4. prolog 程序实际上是一个智能数据库

prolog 的原理就是关系数据库，它是建立在关系数据库的基础上的。在以后的学习中你会发现它和 SQL 数据库查询语言有很多相似之处。使用 prolog 可以很方便的处理数据。

5. 强大的递归功能

在其它的语言中，你也许已经接触过递归程序了。递归是一种非常简洁的方式，它能够有效的解决许多难题。而在 prolog 中，递归的功能得到了充分的体现，你甚至都会感到惊奇，递归居然又如此巨大的能力。

下一步该怎么做

如果你决定下来要学习 prolog 了，那么请继续看这里的教程。你要注意哦，这里是目前全球唯一的详细介绍 prolog 的中文网站。

1. 在学习之前，希望你能够搞到比较好的 prolog 解释器，下一节我将就解释器进行一些讨论。
2. 然后你必须熟练的掌握解释器的使用方法。
3. 然后就可以开始阅读我的教程了。
4. 当你学习完整个教程以后，希望你能够进入人工智能实例环节，那里有更多的、更有用的 prolog 编程方法，和有趣的程序。
5. 如果你想使用 prolog 和其它的语言结合起来，编写让人瞠目结舌的又聪明、又漂亮的程序，你就应该仔细研究 VB+prolog 这一节。

好了，还等什么？ 让我们开始吧。

第 2 章-入门

prolog-使用逻辑编程语言的教程，第二章 - 入门。主要包括探索 Prolog，逻辑编程等。

探索 Prolog

Prolog 在英语中的意思就是 Programming in LOGic（逻辑编程）。它是建立在逻辑学的理论基础之上的， 最初是运用于自然语言的研究领域。然而现在它被广泛的应用在人工智能的研究中，它可以用来建造专家系统、自然语言理解、智能知识库等。同时它对一些通常的应用程序的编写也很有帮助。使用它能够比其他的语言更快速地开发程序，因为它的编程方法更象是使用逻辑的语言来描述程序。

从纯理论的角度来讲，Prolog 是一种令人陶醉的编程语言，但是在这本书中还是着重介绍他的实际使用方法。

进入 Prolog 世界

和其他的语言一样，最好的学习方法是实践。这本书将使用 Prolog 的解释器来向大家介绍几个具体的应用程序的编写过程。

首先你应该拥有一个 Prolog 的解释器，你可以在 Google 中找到它。关于解释器的使用，请参阅相关的使用说明文档，建议使用 amzi prolog 或者 swi prolog 来运行本网站的程序。

逻辑编程

什么叫逻辑编程？也许你还没有一个整体的印象，还是让我们首先来研究一个简单的例子吧。运用经典的逻辑理论，我们可以说“所有的人（person）都属于人类（moral）”，如果用 Prolog 的语言来说就是“对于所有的 X，只要 X 是一个人，它就属于人类。”

```
moral(X):-person(X).
```

同样，我们还可以加入一些简单的事实，比如：苏格拉底（socrates）是一个人。

```
person(socrates).
```

有了这两条逻辑声明，Prolog 就可以判断苏格拉底是不是属于人类。在 Prolog 的 Listener 中键入如下的命令：

?-mortal(socrates).（此句中的'?-'是 Listener 的提示符，本句表示询问苏格拉底是不是属于人类。）

Linstener 将给出答案：

yes

我们还可以询问，“谁属于人类？”

?-mortal(X).

我们会得到如下的答案：

X= socrates

这个简单的例子显示了 Prolog 的一些强大的功能。它能让程序代码更简洁、更容易编写。在多数情况下 Prolog 的程序员不需要关心程序的运行流程，这些都由 Prolog 自动地完成了。

当然，一个完整的程序不能只包括逻辑运算部分，还必须拥有输入输出，乃至用户界面部分。很遗憾，Prolog 在这些方面做得不好，或者说很差。不过它还是提供了一些基本的方法的。下面是上述的程序一个完整的例子。

```
% This is the syntax for comments. % MORTAL - The first illustrative Prolog

program mortal(X) :- person(X).

person(socrates).

person(plato).

person(aristotle).

mortal_report:-

write('Known mortals are:'),nl, mortal(X), write(X),nl,

fail.
```

把这个程序调入 Listener 中，运行 mortal_report.。

```
?- mortal_report.
```

```
Known mortals are:

socrates

plato

aristotle

no
```

以上程序中的一些函数以后还会详细的介绍的。最后的那个 no 表示没有其他的人了。

进入下一章

从下一章起，就开始正式介绍 Prolog 的编程方法了。我将用一个实例来介绍 Prolog，这是一个文字的冒险游戏，你所扮演的角色是一个三岁的小女孩，你想睡觉了，可是没有毛毯（nani）你就不能安心的睡觉。所以你必须在那个大房子中找到你的毛毯，这就是你的任务。这个游戏能够显示出一些 Prolog 的独到之处，不过 Prolog 的功能远不止编个简单的游戏，所以文中还将介绍一些其他的小程序。

第 3 章-事实

prolog-使用逻辑编程语言的教程，第二章 - 事实。主要包括用 Prolog 事实表述数据等内容。

事实 (facts)

注：斜粗体字表示 Prolog 的专有名词

事实（facts）是 prolog 中最简单的谓词（predicate）。它和关系数据库中的记录十分相似。在下一章中我们会把事实作为数据库来搜索。

谓词： Prolog 语言的基本组成元素，可以是一段程序、一个数据类型或者是一种关系。它由谓词名和参数组成。两个名称相同而参数的数目不同的谓词是不同的谓词。

事实的语法结构如下：

pred(arg1, arg2, ... argN).

其中 pred 为谓词的名称。arg1，...为参数，共有 N 个。‘.’是所有的 Prolog 子句的结束符。没有参数的谓词形式如下：

pred.

参数可以是以下四种之一：

- 整数（integer)
- 绝对值小于某一个数的正数或负数。
- 原子（atom）
- 由小写字母开头的字符串。
- 变量（variable）
- 由大写字母或下划线（_）开头。
- 结构（structure）
- 在以后的章节介绍。

不同的 Prolog 还增加了一些其他的数据类型，例如浮点数和字符串等。

Prolog 字符集包括： 大写字母，A-Z；小写字母，a-z；数字，0-9；
+-/\^,.~:.?#$等。

原子通常是字母和数字组成，开头的字符必须是小写字母。例如：

```
hello

twoWordsTogether

x14
```

为了方便阅读，可以使用下划线把单词分开。例如：

```
a_long_atom_name

z_23
```

下面的是不合法的原子，

```
no-embedded-hyphens

123nodigitsatbeginning

Nocapsfirst
```

下划线不能放在最前面

使用单引号扩起来的字符集都是合法的原子。例如：

```
'this-hyphen-is-ok'

'UpperCase'

'embedded blanks'
```

下面的由符号组成的也是合法的原子：

```
>,++
```

变量和原子相似， 但是开头字符四大写字母或是下划线。例如：

X

Input_List

下划线开头的都是变量

Z56

有了这些基本的知识，我们就可以开始编写事实了。事实通常用来储存程序所需的数据。例如，某次商业买卖中的顾客数据。customer/3。（/3 表示 customer 有三个参数）

customer('John Jones', boston, good_credit).

customer('Sally Smith', chicago, good_credit).

必须使用单引号把顾客名引起来，因为它们是由大写字母开头的，并且中间有空格。

再看一个例子，视窗系统使用事实储存不同的窗口信息。在这个例子中参数有窗口名称和窗口的位置坐标。

window(main, 2, 2, 20, 72).

window(errors, 15, 40, 20, 78).

某个医疗专家系统可能有如下的疾病数据库。

disease(plague, infectious). {疾病（瘟疫，有传染性）}

Prolog 的解释器提供了动态储存事实和规则的方法，并且也提供了访问它们的方法。数据库的更新是通过运行‘consult’或‘reconsult’命令。我们也可以直接在解释器中输入谓词，但是这些谓词不会被储存到硬盘上。

寻找 Nani

下面我们正式开始“寻找 Nani”游戏的编写。我们从定义基本的事实开始，这些事实是本游戏的基本的数据库。它们包括：

房间和它们的联系

物体和它们的位置

物体的属性

玩家在游戏开始时的位置

“寻找 Nani”游戏的的房间格局

首先我们使用 room/1 谓词定义房间，一共有五条子句，它们都是事实，如图 2.1。

```
room(kitchen).
```

```
room(office).

room(hall).

room('dining room').

room(cellar).
```

我们使用具有两个参数的谓词来定义物体的位置。第一个参数代表物体的名称，第二个参数表示物体的位置。开始时，我们加入如下的物体。

```
location(desk, office).

location(apple, kitchen).

location(flashlight, desk).

location('washing machine', cellar).

location(nani, 'washing machine').

location(broccoli, kitchen).

location(crackers, kitchen).

location(computer, office).
```

注意：我们定义的那些符号，例如：kitchen、desk 等对于我们是有意义的，可是它们对于 Prolog 是没有任何意义的，完全可以使用任何符号来表示房间的名称。

谓词 location/2 的意思是“第一个参数所代表的物体位于第二个参数所代表的物体中”。Prolog 能够区别 location(sink, kitchen)和 location(kitchen, sink)。因此，参数的顺序是我们定义事实时需要考虑的一个重要问题。

下面我们来表达房间的联系。使用 door/2 来表示两个房间有门相连，这里遇到了一个小小的困难：

```
door(office, hall).
```

我们想要表达的意思是，office 和 hall 之间有一个门。可是由于 Prolog 能够区分 door(office, hall)和 door(hall, office)， 所以如果我们想要表达一种双向的联系，就必须把每种联系都定义一遍。

```
door(office, hall).
```

door(hall, office).

参数的顺序对定义物体的位置有帮助，可是在定义房间的联系时却带来了麻烦。我们不得不把每个房门都定义两次！

在这一章里，只定义单向的门，以后会很好地解决此问题的。

```
door(office, hall).

door(kitchen, office).

door(hall, 'dining room').

door(kitchen, cellar).

door('dining room', kitchen).
```

下面定义某些物体的属性，

```
edible(apple).

edible(crackers).



tastes_yucky(broccoli).
```

最后，定义手电筒（由于是晚上，玩家必须想找到手电筒，并打开它才能到那些关了灯的房间）的状态和玩家的初始位置。

```
turned_off(flashlight).

here(kitchen).
```

好了，到此你应该学会了如何使用 Prolog 的事实来表达数据了。

第 4 章-简单查询

prolog-使用逻辑编程语言的教程，第四章 - 简单查询。主要包括简单查询及查询的工作原理等。

现在我们的游戏中已经有了一些事实，使用 Prolog 的解释器调入此程序后，我们就可以对这些事实进行查询了。本章和下一章中的 Prolog 程序只包括事实，我们要学会如何对这些事实进行查询。

Prolog 的查询工作是靠模式匹配完成的。查询的模板叫做目标(goal)。如果有某个事实与目标匹配，那么查询就成功了，Prolog 的解释器会回显'yes.'。如果没有匹配的事实，查询就失败了，解释器回显'no.'。

我们把 Prolog 的模式匹配工作叫做联合(unification)。当数据库中只包括事实时，以下三个条件是使联合成功的必要条件。

目标谓词名与数据库中的某个谓词名相同。

这两个谓词的参数数目相同。

所有的参数也相同。

在介绍查询之前，让我们回顾一下上一章所编写的 Prolog 程序。

```
room(kitchen).

room(office).

room(hall).

room('dining room').

room(cellar).

door(office, hall).

door(kitchen, office).

door(hall, 'dining room').

door(kitchen, cellar).

door('dining room', kitchen).
```

```
location(desk, office).
location(apple, kitchen).
location(flashlight, desk).
location('washing machine', cellar).
location(nani, 'washing machine').
location(broccoli, kitchen).
location(crackers, kitchen).
location(computer, office).

edible(apple).
edible(crackers).
tastes_yucky(broccoli).
here(kitchen).
```

以上是我们的“寻找 Nani”中的所有事实。把这段程序调入 Prolog 解释器中后就可以开始进行查询了。

我们的第一个问题是：office 在本游戏中是不是一个房间。

?-room(office). {?-是解释器的提示符}

yes.

Prolog 回答 yes，因为它在数据库中找到了 room(office).这个事实。我们继续问：有没有 attic 这个房间。

?-room(attic).

no.

Prolog 回答 no，因为它在数据库中找不到 room(attic).这个事实。同样我们还可以进行如下的询问。

```
?- location(apple, kitchen).

yes

?- location(kitchen, apple).

no
```

你看 Prolog 懂我们的意思呢，它知道苹果在厨房里，并且知道厨房不在苹果里。但是下面的询问就出问题了。

```
?- door(office, hall).

yes

?- door(hall, office).

no
```

由于我们定义的门是单方向的，结果遇到了麻烦。

在查询目标中我们还可以使用 Prolog 的变量。这种变量和其他语言中的不同。叫它逻辑变量更合适一点。变量可以代替目标中的一些参数。

变量给联合操作带来了新的意义。以前联合操作只有在谓词名和参数都相同时才能成功。但是引入了变量之后，变量可以和任何的条目匹配。

当联合成功之后，变量的值将和它所匹配的条目的值相同。这叫做变量的绑定(binding)。当带变量的目标成功的和数据库中的事实匹配之后，Prolog 将返回变量绑定的值。

由于变量可能和多个条目匹配,Prolog 允许你察看其他的绑定值。在每次 Prolog 的回答后输入“；”，可以让 Prolog 继续查询。下面的例子可以找到所有的房间。“；”是用户输入的。

```
?- room(X).

X = kitchen ;

X = office ;

X = hall ;
```

```
X = 'dining room' ;

X = cellar ;

no
```

最后的 no 表示找不到更多的答案了。

下面我们想看看 kitchen 中都有些什么。（变量以大写字母开始）

```
?- location(Thing, kitchen).

Thing = apple ;

Thing = broccoli ;

Thing = crackers ;

no
```

我们还可以使用两个变量来查询所有的物体及其位置。

```
?- location(Thing, Place).

Thing = desk

Place = office ;


Thing = apple

Place = kitchen ;


Thing = flashlight

Place = desk ;

...

no
```

查询的工作原理

当 Prolog 试图与某一个目标匹配时，例如：location/2，它就在数据库中搜寻所有用 location/2 定义的子句，当找到一条与目标匹配时，它就为这条子句作上记号。当用户需要更多的答案时，它就从那条作了记号的子句开始向下查询。

我们来看一个例子，用户询问：location（X，kitchen）.。Prolog 找到数据库中的第一条 location/2 子句，并与目标比较。

目标 location(X, kitchen)

子句#1 location(desk, office)

匹配失败，因为第二个参数不同，一个是 kitchen，一个是 office。于是 Prolog 继续比较第二个子句。

目标 location(X, kitchen)

子句#2 location(apple, kitchen)

这回匹配成功，而变量 X 的值就被绑定成了 apple。

?- location(X, kitchen).

X = apple

如果用户输入分号(;)，Prolog 就开始寻找其他的答案。首先它必须释放（unbinds）变量 X。然后从上一次成功的位置的下一条子句开始继续搜索。这个过程叫做回溯（backtracking）。在本例中就是第三条子句。

目标 location(X, kitchen)

子句#3 location(flashlight, desk)

匹配失败，直到第六条子句时匹配又成功了 。

目标 location(X, kitchen)

子句#6 location(broccoli, kitchen)

结果变量 X 又被绑定为 broccoli，解释器显示：

X = broccoli ;

再度输入分号，X 又被解放，开始新的搜索。又找到了：

X = crackers ;

这回再没有新的子句能够匹配了，于是 Prolog 回答 no，表示最后一次搜索失败了。

no

要想了解 Prolog 的运行顺序，最好的方法就是单步调试程序，不过在此之前，还是让我们加深一下对目标的认识吧。

Prolog 的目标有四个端口用来控制运行的流程：调用（call）、退出（exit）、重试（redo）以及失败（fail）。一开始使用 Call 端口进入目标，如果匹配成功就到了 exit 端口，如果失败就到了 fail 端口，如果用户输入分号，则又从 redo 端口进入目标。下图表示了目标和它的四个端口。

每个端口的功能如下：

- call 开始使用目标搜寻子句。
- exit 目标匹配成功，在成功的子句上作记号，并绑定变量。
- redo 试图重新满足目标，首先释放变量，并从上次的记号开始搜索。
- fail 表示再找不到更多的满足目标的子句了。

下面列出了调试 location(X,kitchen).时的情况。括号中的数字表示当前正在考虑的子句。

```
?- location(X, kitchen).

CALL: - location(X, kitchen)

EXIT:(2) location(apple, kitchen)

X = apple;

REDO: location(X, kitchen)

EXIT:(6) location(broccoli, kitchen)

X = broccoli ;

REDO: location(X, kitchen)

EXIT:(7) location(crackers, kitchen)
```

```
X = crackers ;

FAIL - location(X, kitchen)

no
```

在 Prolog 的解释器中输入，

?- debug.

就可以开始调试你的程序了。

第 5 章-混合查询

prolog-使用逻辑编程语言的教程，第五章 - 混合查询。主要包括复杂的混合查询和内部谓词等。

混合查询

我们可以把简单的查询连接起来，组成一些较复杂的查询。例如，如果我们想知道厨房里能吃的东西，就可以向 Prolog 进行如下的询问。

```
?- location(X, kitchen), edible(X).
```

简单的查询只有一个目标，而混合查询可以把这些目标连接起来，从而进行较为复杂的查询。上面的连接符号','是并且的意思。

上面的式子用语言来描述就是“寻找满足条件的 X，条件是：X 在厨房里，并且 X 能吃。”如果某个变量在询问中多次出现，则此变量在所有出现的位置都必须绑定为相同的值。所以上面的查询只有找到某一个 X 的值，使得两个目标都成立时，才算查询成功。

每次查询所使用的变量都是局部的变量，它只在本查询中有意义，所以当我们进行了如下的查询后，

```
?- location(X, kitchen), edible(X).

X = apple ;

X = crackers ;

no
```

查询结果中没有 broccoli（椰菜），因为我们没有把它定义为可吃的东西。此后，还可以用 X 进行其他的查询。

```
?- room(X).
X = kitchen ;
X = office ;
X = hall ;
...;
no
```

除了使用逻辑的方法理解混合查询外，还可以通过分析程序的运行步骤来理解。用程序的语言来说就是“首先找到一样位于厨房的东西，然后判断它能否食用，如果不能，就到厨房里找下一样东西，再判断能否食用。一直如此重复，直到找到答案或把厨房的东西全部查完为止。”

请参照下图来理解。

调用查询后，程序将按照下面的步骤运行，请参照上图来理解。

搜索第一个目标，如果成功转到 2，如果失败则回答’no’，查询结束。

搜索第二个目标，如果成功转到 3，如果失败转到 1。

把绑定的变量的值输出。用户输入’;’后转到 2。

上面的例子中只有一个变量，下面我们再来看一个有两个变量的例子。

```
?- door(kitchen, R), location(T,R).

R = office

T = desk ;



R = office

T = computer ;
```

```
R = cellar

T = 'washing machine' ;


no
```

上面的查询用逻辑的语言来解释就是：“找房间 R，使得从厨房到房间 R 有门相连，并且把房间 R 中的物品 T(这里是房间 R 的所有物品）也找出来。”

下面是此查询的单步运行过程。

```
Goal: door(kitchen, R), location(T,R)


1 CALL door(kitchen, R)

1 EXIT (2) door(kitchen, office)

2 CALL location(T, office)

2 EXIT (1) location(desk, office)

R = office

T = desk ;


2 REDO location(T, office)

2 EXIT (8) location(computer, office)

R = office

T = computer ;
```

```
2 REDO location(T, office)

2 FAIL location(T, office)

1 REDO door(kitchen, R)

1 EXIT (4) door(kitchen, cellar)

2 CALL location(T, cellar)

2 EXIT (4) location('washing machine', cellar)

R = cellar

T = 'washing machine' ;


2 REDO location(T, cellar)

2 FAIL location(T, cellar)

1 REDO door(kitchen, R)

1 FAIL door(kitchen, R)

no
```

内部谓词

讲了这么多了，我们还只是用到了 Prolog 的一些语法，完全没有使用 Prolog 提供的一些内部的函数，我把这些内部函数称为内部谓词。和其他的程序语言一样，Prolog 也提供了一些基本的输入输出函数，下面我们要编写一个较复杂的查询，它能够找到所有厨房里能够吃的东西，并把它们列出来。而不是像以前那样需要人工输入';'。

要想完成上面的任务，我们首先必须了解内部谓词的概念。内部谓词是指已经在 Prolog 中事先定义好的谓词。在内存中的动态数据库中是没有内部谓词的子句的。当解释器遇到了内部谓词的目标，它就直接调用事先编好的程序。

内部谓词一般所完成的工作都是与逻辑程序无关的，例如输入输出的谓词。所以我们可以把这些谓词叫做非逻辑谓词。

但是这些谓词也可以作为 Prolog 的目标，所以它们也必须拥有和逻辑谓词相同的四个端口：Call、Fail、Redo 和 Exit。

下面介绍几个常用的输出谓词。

write/1

此谓词被调用时永远是成功的，并且它可以把它的参数作为字符串输出到屏幕上。当回溯时，它永远是失败，所以回溯是不会把已经写到屏幕上的字符又给删除的。

nl/0

此谓词没有参数，和 write 一样，从 Call 端口调用时总是成功的，从 Redo 端口回溯时总是失败的，它的作用是在屏幕上输出一个回车符。

tab/1

此谓词的参数是一个整数，它的作用是输出 n 个空格，n 为它的参数。其控制流程与上面两个相同。

下图是一般情况下的 Prolog 目标的内部流程控制示意图。我们将使用此图和内部谓词的流程控制图相比较。

上图中左上角的菱形方块表示从 Call 端口进入目标时所进行的处理。它从某谓词的第一个子句开始搜索，如果匹配成功就到 Exit 端口，如果没有找到任何一个子句与目标匹配就转到 Fail 端口。

右下角的方块表示从 Redo 端口进入目标时所进行的处理，从最近一次成功的子句开始向下搜索，如果匹配成功就转到 Exit 端口，如果没有找个更多的子句满足目标就转到 Fail 端口。

I/O 谓词的流程控制和上述的不同，它不会改变流程的方向，如果流程从它的左边进入，就会从它的右边流出；而如果从它的右边进入，则会从它的左边流出。请参考下图理解。

I/O 谓词不会改变变量的值，但是它们可以把变量的值输出。

还有一个专门引起回溯的内部谓词 fail/0，从它的名字不难看出，它的调用永远是失败的。如果 fail/0 从左边得到控制权，则它立即把控制权再传回到左边。它不会从右边得到控制，因为没法通过 fail/0 把控制权传到右侧。它的内部流程控制如下：

以前我们是靠使用';'来进入目标的 Redo 端口的，并且变量的值的输出是靠解释器完成的。现在有了上面几个内部谓词，我们就可以靠 I/O 谓词来显示变量的值，靠 fail 谓词来引起自动的回溯。

下面是此查询语句及其运行结果。

```
?- location(X, kitchen), write(X) ,nl, fail.

apple

broccoli

crackers

no
```

下面是此查询的流程图。

下面是此查询的单步调试过程。

```
Goal: location(X, kitchen), write(X), nl, fail.

1 CALL location(X, kitchen)

1 EXIT (2) location(apple, kitchen)

2 CALL write(apple)

apple

2 EXIT write(apple)

3 CALL nl



3 EXIT nl

4 CALL fail

4 FAIL fail

3 REDO nl

3 FAIL nl

2 REDO write(apple)

2 FAIL write(apple)

1 REDO location(X, kitchen)

1 EXIT (6) location(broccoli, kitchen)

2 CALL write(broccoli)

broccoli

2 EXIT write(broccoli)

3 CALL nl
```

```
3 EXIT nl

4 CALL fail

4 FAIL fail

3 REDO nl

3 FAIL nl

2 REDO write(broccoli)

2 FAIL write(broccoli)

1 REDO location(X, kitchen)

1 EXIT (7) location(crackers, kitchen)

2 CALL write(crackers) crackers

2 EXIT write(crackers)

3 CALL nl



3 EXIT nl

4 CALL fail

4 FAIL fail

3 REDO nl

3 FAIL nl

2 REDO write(crackers)

2 FAIL write(crackers)

1 REDO location(X, kitchen)

1 FAIL location(X, kitchen)

no
```

下面请你分析一下，

```
?- door(kitchen, R), write(R), nl, location(T,R), tab(3), write(T), nl, fail.
```

的输出的结果是什么呢？

第 6 章-规则

prolog-使用逻辑编程语言的教程，第六章 - 规则。主要包括规则定义，规则的工作原理和使用规则等。

规则

前面我们已经说过，谓词是使用一系列的子句来定义的。以前我们所学习的子句是事实，现在让我们来看看规则吧。规则的实质就是储存起来的查询。它的语法如下：

```
head :- body
```

其中，

- head 是谓词的定义部分，与事实一样，也包括谓词名和谓词的参数说明。
- :- 连接符，一般可以读作‘如果’。
- body 一个或多个目标，与查询相同。

举个例子，上一章中的混合查询--找到能吃的东西和它所在的房间，可以使用如下的规则保存，规则名为 where_food/2。

```
where_food(X,Y) :- location(X,Y), edible(X).
```

用语言来描述就是“在房间 Y 中有可食物 X 的条件是：X 在 Y 房间中，并且 X 可食。”

我们现在可以直接使用此规则来找到房间中可食的物品。

```
?- where_food(X, kitchen).

X = apple ;
```

```
X = crackers ;

no


?- where_food(Thing, 'dining room').

no
```

它也可以用来判断，

```
?- where_food(apple, kitchen).

yes
```

或者通过它找出所有的可食物及其位置，

```
?- where_food(Thing, Room).

Thing = apple

Room = kitchen ;


Thing = crackers

Room = kitchen ;

no
```

我们可以使用多个事实来定义一个谓词，同样我们也可以用多个规则来定义一个谓词。例如，如果想让 Prolog 知道 broccoli（椰菜）也是可食物，我们可以如下定义 where_food/2 规则。

```
where_food(X,Y) :- location(X,Y), edible(X).

where_food(X,Y) :- location(X,Y), tastes_yucky(X).
```

在以前的事实中我们没有把broccoli定义为edible,即没有edible(broccoli).这个事实，所以单靠where_food的第一个子句是不能找出broccoli的，但是我们曾在事实中定义过：tastes_yucky(broccoli).{不好吃(椰菜).}，所以如果加入第二个子句，Prolog就可以知道broccoli也是food（食物）了。下面是它的运行结果。

```
?- where_food(X, kitchen).

X = apple ;

X = crackers ;

X = broccoli ;

no
```

规则的工作原理

到现在为止，我们所知道的Prolog所搜索的子句只有事实。下面我们来看看Prolog是如何搜索规则的。

首先，Prolog将把目标和规则的子句的头部（head）进行匹配，如果匹配成功，Prolog就把此规则的body部分作为新的目标进行搜索。

实际上规则就是多层的询问。第一层由原始的目标组成，从下一层开始就是由与第一层的目标相匹配的规则的Body中的子目标组成。（这句话有点难理解，请参照下面图来分析）

每一层还可以有子目标，理论上来讲，这种目标的嵌套可以是无穷的。但是由于计算机的硬件限制，子目标只可能有有限次嵌套。

下图显示了这种目标嵌套的流程图，请你注意第一层的第三个目标是如何把控制权回溯到第二层的子目标中的。

在这个例子中，第一层的中间的那个目标的结果依赖于第二层的目标的结果。此目标会把程序的控制权传给他的子目标。

下面我们详细地分析一下 Prolog 在匹配有规则的子句时是如何工作的。请注意用‘-’分隔的两个数字，第一个数字代表当前的目标级数，第二个数字代表当前目标层中正在匹配的目标的序号。例如：

2-1 EXIT (7) location(crackers, kitchen)

表示第二层的第一个目标的 EXIT 过程。

我们的询问如下

?- where_food(X, kitchen).

首先我们寻找有 where_food/2 的子句.

1-1 CALL where_food(X, kitchen)

与第一个子句的头匹配

1-1 try (1) where_food(X, kitchen) ;第一个 where_food/2 的子句与目标匹配。

于是第一个子句的 Body 将变为新的目标。

2-1 CALL location(X, kitchen)

从现在起的运行过程就和我们以前一样了。

2-1 EXIT (2) location(apple, kitchen)

2-2 CALL edible(apple)

2-2 EXIT (1) edible(apple)

由于 Body 的所有目标都成功了，所以第一层的目标也就成功了。

1-1 EXIT (1) where_food(apple, kitchen)

X = apple ;

第一层的回溯过程使得又重新进入了第二层的目标。

1-1 REDO where_food(X, kitchen)

2-2 REDO edible(apple)

2-2 FAIL edible(apple)

2-1 REDO location(X, kitchen)

2-1 EXIT (6) location(broccoli, kitchen)

2-2 CALL edible(broccoli)

2-2 FAIL edible(broccoli)

2-1 REDO location(X, kitchen)

2-1 EXIT (7) location(crackers, kitchen)

2-2 CALL edible(crackers)

2-2 EXIT (2) edible(crackers)

1-1 EXIT (1) where_food(crackers, kitchen)

X = crackers ;

下面就没有更多的答案了，于是第一层的目标失败。

2-2 REDO edible(crackers)

2-2 FAIL edible(crackers)

2-1 REDO location(X, kitchen)

2-1 FAIL location(X, kitchen)

下面 Prolog 开始寻找另外的子句，看看它们的头部（head）能否与目标匹配。在此例中，where_food/2 的第二个子句也可以与询问匹配。

1-1 REDO where_food(X, kitchen)

Prolog 又开始试图匹配第二个子句的 Body 中的目标。

1-1 try (2) where_food(X, kitchen) ;第二个 where_food/2 的子句与目标匹配。

下面将找到不好吃的椰菜。即 tastes_yucky 的 broccoli.

2-1 CALL location(X, kitchen)

2-1 EXIT (2) location(apple, kitchen)

2-2 CALL tastes_yucky(apple)

2-2 FAIL tastes_yucky(apple)

2-1 REDO location(X, kitchen)

2-1 EXIT (6) location(broccoli, kitchen)

2-2 CALL tastes_yucky(broccoli)

2-2 EXIT (1) tastes_yucky(broccoli)

1-1 EXIT (2) where_food(broccoli, kitchen)

X = broccoli ;

回溯过程将让 Prolog 寻找另外的 where_food/2 的子句。但是，这次它没有找到。

2-2 REDO tastes_yucky(broccoli)

2-2 FAIL tastes_yucky(broccoli)

2-1 REDO location(X,kitchen)

2-1 EXIT (7) location(crackers, kitchen)

2-2 CALL tastes_yucky(crackers)

2-2 FAIL tastes_yucky(crackers)

2-2 REDO location(X, kitchen)

2-2 FAIL location(X, kitchen)

1-1 REDO where_food(X, kitchen) ;没有找到更多的 where_food/2 的子句了。

1-1 FAIL where_food(X, kitchen)

no

在询问的不同层的目标中，即是相同的变量名称也是不同的变量，因为它们都是局部变量。这于其他语言中的局部变量是差不多的。

我们再来分析一下上面的那个例子吧。

where_food(X,Y) :- location(X,Y), edible(X).

查询的目标是：

?- where_food(X1, kitchen)

第一个子句的头是：

where_food(X2, Y2)

目标和子句的头部匹配，在 Prolog 中如果变量和原子匹配，那么变量就绑定为此原子的值。如果两个变量进行了匹配，那么这两个变量将同时绑定为一个内部

变量。此后，这两个变量中只要有一个绑定为了某个原子的值，另外一个变量也会同时绑定为此值。所以上面的匹配操作将有如下的绑定。

X1 = _01 ;01 为 Prolog 的内部变量。

X2 = _01

Y2 = kitchen

于是当上述的匹配操作完成后，规则 where_food/2 的 body 将变成如下的查询：

location(_01, kitchen), edible(_01).

当内部变量取某值时，例如'apple'，X1 和 X2 将同时绑定为此值，这是 Prolog 变量和其他语言的变量的基本的区别。如果你学过 C 语言，容易看出，实际上 X1 和 X2 都是指针变量，当它们没有绑定值时，它们的值为 NULL，一旦绑定，它们就会指向某个具体的位置，上例中它们同时指向了_01 这个变量，其实_01 变量还是个指针，直到最后某个指针指向了具体的值，那么所有的指针变量就都被绑定成了此值。

使用规则

使用规则我们可以很容易的解决单向门的问题。我们可以再定义有两个子句的谓词来描述这种双向的联系。此谓词为 connect/2。

```
connect(X,Y) :- door(X,Y).

connect(X,Y) :- door(Y,X).
```

它代表的意思是“房间 X 和 Y 相连的条件是：从 X 到 Y 有扇门，或者从 Y 到 X 有扇门”。请注意此处的或者， 为了描述这种或者的关系我们可以为某个谓词定义多个子句。

```
?- connect(kitchen, office).

yes


?- connect(office, kitchen).

yes
```

我们还可以让 Prolog 列出所有相连的房间。

```
?- connect(X,Y).

X = office

Y = hall ;


X = kitchen

Y = office ;


...

X = hall

Y = office ;


X = office

Y = kitchen ;

...
```

使用我们现在所具有的知识，我们可以为“搜索 Nani”加入更多的谓词。首先我们定义 look/0，它能够显示玩家所在的房间，以及此房间中的物品和所有的出口。

先定义 list_things/1，它能够列出某个房间中的物品。

```
list_things(Place) :-

location(X, Place),

tab(2),

write(X),

nl,

fail.
```

它和上一章中的最后一个例子差不多。我们可以如下使用它。

```
  ?- list_things(kitchen).

apple

broccoli

crackers

no
```

这地方有一个小问题，它虽然把所有的东西都列出来了，但是最后那个 no 不太好看，并且如果我们把它和其他的规则连起来用时麻烦就更大了，因为此规则的最终结果都是 fail。实际上它是我们扩充的 I/O 谓词，所以它应该总是成功的。我们可以很容易的解决这个问题。

```
list_things(Place) :-

location(X, Place),

```

```
tab(2),

write(X),

nl, fail.

list_things(AnyPlace).
```

如上所示，加入 list_things(AnyPlace)子句后就可以解决了，第一个 list_things/1 的子句把所有的物品列出，并且失败，而由于第二个子句永远是成功的，所以 list_things/1 也将是成功的。AnyPlace 变量的值我们并不关心，所以我们可以使用无名变量‘_’来代替它。

```
list_things(_).
```

下面我们来编写 list_connections/1，它能够列出与某个房间相连的所有房间。

```
list_connections(Place)

:- connect(Place, X),

tab(2),

write(X),

nl,

fail.


list_connections(_).
```

我们来试试功能，

```
?- list_connections(hall).
```

```
dining

room

office

yes
```

终于可以来编写 look/0 了，

```
look :-

here(Place),

write('You are in the '),

write(Place),

nl,

write('You can see:'),

nl,

list_things(Place),

write('You can go to:'),

nl,

list_connections(Place).
```

在我们定义的事实中有 here(kitchen).它代表玩家所在的位置。以后我们将学习如何改变此事实。现在来试是功能吧，

```
?- look.

You are in the kitchen

You can see:
```

```
apple

broccoli

crackers

You can go to:

office

cellar

dining

room

yes
```

好了到此，我们已经学会了 Prolog 的基本编程方法，下一章将总结一下，并再举几个例子，此后我们将进入较深的学习。

第 7 章-小结

prolog-使用逻辑编程语言的教程，第七章 - 小结。主要是对前面六章内容的一个小结。

小结

到现在为止，我们已经对 Prolog 有了一个基本的了解，现在有必要对我们所学过的知识做一个系统的总结。

- Prolog 的程序是由一系列的事实和规则组成的数据库。
- 规则之间的调用是通过联合操作完成的，Prolog 能够自动的完成模式匹配。
- 规则还可以调用内部谓词，例如 write/1。
- 我们可以在 Prolog 的解释器中单独地对规则进行查询（调用）。

在 Prolog 的程序的运行流程方面我有了如下的认识：

- 规则的运行是通过 Prolog 内建的回溯功能实现的。
- 我们可以使用内部谓词 fail 来强制实现回溯。
- 我们也可以通过加入一条参数为伪变量（下划线）无 Body 部分的子句，来实现强制让谓词成功。

我们还学习了，

- 数据库中的事实代替了一般语言中的数据结构。
- 回溯功能能够完成一般语言中的循环操作。
- 而通过模式匹配能够完成一般语言中的判断操作。
- 规则能够被单独地调试，它和一般语言中的模块相对应。
- 而规则之间的调用和一般语言中的函数的调用类似。

有了以上的知识，我们还可以编写出一些让其它语言的程序员吃惊的小程序。下面就举一个分析家谱的程序。

假如我们把家族成员之间的父子关系和夫妻关系，以及成员的性别属性定义为基本的事实数据库，我们就可以编出许多规则来判断其他的亲戚关系了。

例如我们有如下的数据库：

```
father(a,b).

father(a,d).

father(a,t).

father(b,c).



wife(aw,a).

wife(bw,b).



male(t).

female(d).

male(c).
```

father(a,b).表示 a 是 b 的父亲。

wife(aw,a). 表示 aw 是 a 的妻子。

male(t).表示 b 是男性。

female(d).表示 d 是女性。

上面我们并没有定义 a、b、aw、bw 的性别。 因为通过他们和其他人的关系我们可以很容易地确定他们的性别。不过要想让 Prolog 知道他们的性别我们就要定义如下的规则。

```
male(X):-father(X,_).

female(X):-wife(X,_).
```

上面的 male/1 和 female/1 的谓词名称和事实的名称相同，这并不是什么特别的情况，你可以把所有定义相同的谓词的子句之间的关系想象“或者”的关系。也就是说：t 和 d 是男性，或者如果 X 是其他人的父亲，则它也是男性。在判断性别时，我们并不关心此人是谁的父亲，所以后面一个变量用“_”代替了。

好了，假如有如下的询问：

```
?-male(t).

yes.



?-male(a).

yes.



?-male(X).

X=t;

X=c;

X=a;

X=a;

X=a;

X=b;

no.
```

最后一个询问，它虽然把所有的男性找了出来，可是它把 a 找了三次，原因很简单，因为我们有三个 father/2 的子句都包含 a，好像不太理想，不过现在只能将就一下了，当我们学习了更多的知识后，就好解决了。

下面我们定义一些其他的亲戚关系的规则。你大概一看就能够理解。例如：X 和 Y 是兄弟的条件是: X 和 Y 有相同的父亲{father(Z,X),father(Z,Y)}，并且他们都是男性{male(X),male(Y)}，最后由于 X 和 Y 可以取相同的值，所以我们不得不加上一条 X 和 Y 不是同一个人{X\=Y}。

```
grandfather(X,Y):-father(X,Z),father(Z,Y).

mother(X,Y):-wife(X,Z),father(Z,Y).

brother(X,Y):-father(Z,X),father(Z,Y),male(X),male(Y),X\=Y.
```

当然我们还可以加入更复杂一点的规则，

```
uncle(X,Y):-brother(X,Z),father(Z,Y).
```

这个叔伯的规则 uncle/2 调用了前面的规则 brother/2。

这里只是简单回顾一下前面所学习的知识，所以这个家族程序虽然可以使用，但是却极不完善。例如：它会把某一答案重复多次，还不能描述没有小孩的丈夫的性别。 我们这样改一下会更好一点：male(X):-wife(_,X)。因此，规则的定义是多种多样的，到底哪种更好、哪种更快，这就是我们以后所要研究的问题之一了。

第 8 章-算术

prolog-使用逻辑编程语言的教程，第八章 - 算术。主要包括如何使用内部谓词 is 来计算数学表达式等。

Prolog 中也有一些能够进行数学计算的功能，但是数学计算是不好用逻辑的事物来描述的。因此计算一个数学表达式的方法和我们以前所学习的模式匹配有很大的区别。因此，Prolog 专门提供了内部谓词 is 来计算数学表达式。其语法形式如下：

X is〈数学表达式〉

变量 X 将被赋值为表达式的值，在回溯时不赋值。数学表达式的形式和其他的语言相同。下面是使用 Prolog 计算的一些例子。

```
?- X is 2 + 2.

X = 4



?- X is 3 * 4 + 2.

X = 14
```

我们还可以使用括号，

```
?- X is 3 * (4 + 2).

X = 18



?- X is (8 / 4) / 2.

X = 1
```

除了 is 以外，Prolog 还提供了一些用来比较大小的操作符。

```
  X > Y

X < Y

X >= Y

X =< Y
```

请注意>=和=<，它们的符号顺序是不能颠倒的。下面是一些例子，

```
?- 4 > 3.

yes



?- 4 < 3.

no



?- X is 2 + 2, X > 3.

X = 4



?- X is 2 + 2, 3 >= X.

no



?- 3+4 —> 3*2.

yes
```

我们可以在规则中使用这些符号，例如，

```
c_to_f(C,F) :- F is C * 9 / 5 + 32.

freezing(F) :- F =< 32.
```

c_to_f/2 把摄氏温度转换为华氏温度，freezing 判断华氏温度的冰点。下面是使用这些谓词的例子。

```
?- c_to_f(100,X).

X = 212

yes



?- freezing(15).

yes



?- freezing(45).

no
```

第 9 章-数据管理

prolog-使用逻辑编程语言的教程，第九章 - 数据管理。主要包括动态地控制内存中的子句等。

Prolog 的程序就是谓词的数据库，我们通常把这些谓词的子句写入 Prolog 的程序中的。在运行 Prolog 时，解释器首先把所有的子句调入到内存中。所以这些写在程序中的子句都是固定不变的。那么有没有办法动态地控制内存中的子句呢？Prolog 提供了这方面的功能。这就意味着，Prolog 程序在运行过程中，还能够改变它自己。它使用一些内部谓词来完成这个功能。最重要的几个谓词如下：

asserta(X)

把子句 X 当作此子句的谓词的第一个子句加入到动态数据库中。它和 I/O 内部谓词的流程控制相同。回溯是失败，并且不会取消它所完成的工作。例如：如果内存中已经有了下面的几个事实：

```
people(a).

people(b).

people(c).
```

如果运行了 asserta(people(d))之后，内存中的 people/1 的子句就变成了下面这个样子：

```
people(d).

people(a).

people(b).

people(c).
```

asserta(X)

和 asserta/1 的功能类似，只不过它把 X 子句追加为最后一个子句。

retract(X)

把子句 X 从动态数据库中删除。此操作也是永久性的，也就是说回溯的时候不能撤销此操作。

* 在 swi prolog 中需要对动态操作的谓词名进行声明，例如前面如果希望能够动态修改 people/1 的子句，需要在程序最前面运行：

```
:-dynamic people/1.
```

能够动态的修改数据库显然是很重要的。它有助于我们完成“寻找 Nani”。使用这些谓词，我们可以很方便地改变玩家和物体的位置。

下面我们来设计 goto/1 这个谓词，它能够把玩家从一个房间移到另一个房间。我们采取从顶向下的设计方法，这和我们设计 look/0 时的方法不同。

当玩家键入了 goto 命令之后，首先判断他能否去他想去的位置，如果可以，则移动到此位置，并把此位置的情况告诉玩家。

```
goto(Place):- can_go(Place), move(Place), look.
```

下面来一步一步地完成这些还没定义的谓词。

玩家所能够去的房间的条件是：此房间和玩家所在的房间是相通的，即：

```
can_go(Place):- here(X), connect(X, Place).
```

我们可以马上测试一下，（假定玩家在厨房）

```
?- can_go(office).

yes



?- can_go(hall).

no
```

现在 can_go/1 已经可以工作了，但是如果它在失败时能够给出一条消息就很好了。所以还需要另外增加一条子句，如果第一条子句失败，也就是说不能去那个房间时，第二个子句将显示一条消息。

```
can_go(Place):- here(X), connect(X, Place).

can_go(Place):- write('You can''t get there from here.'), nl, fail.
```

注意第二条子句最后的那个 fail，因为当目标与第二条子句匹配时，表示不能去此房间，所以它应该返回 fail。这次的运行结果比上次要好多了。

```
?- can_go(hall).

You can't get there from here.

no
```

下面再来设计 move/1 谓词,它必须能够动态的修改数据库中的 here 谓词的子句。首先把玩家的旧位置的数据删除，再加上新位置的数据。

```
move(Place):- retract(here(X)), asserta(here(Place)).
```

现在我们可以使用 goto/1 在游戏的所有房间里走动了。

```
?- goto(office).

You are in the office

You can see:

desk

computer

You can go to:

hall

kitchen

yes


?- goto(hall).

You are in the hall

You can see:

You can go to:

dining

room

office

yes


?- goto(kitchen).
```

```
You can't get there from here.

no
```

好像有点游戏的味道了。:)

下面开始编写 take 和 put 谓词，使用这两个谓词，我们可以拿取或丢弃游戏中的物品。使用 have/1 谓词来储存玩加身上所携带的物品，一开始，玩家身上没有物品，所以我们没有在程序的事实中定义 have/1 谓词。

```
take(X):- can_take(X), take_object(X).
```

其中 can_take(X)的设计方法与 can_go/1 相同。

```
can_take(Thing) :- here(Place), location(Thing, Place).

can_take(Thing) :-

write('There is no '), write(Thing), write(' here.'), nl, fail.
```

take_object/1 与 move/1 类似，它首先删除一条 location/1 的子句，然后添加一条 have/1 的子句。这反映出了物品从其所在位置移到玩家身上的过程。

```
take_object(X):-

retract(location(X,_)), asserta(have(X)), write('taken'), nl.
```

正如我们所看到的那样，Prolog 子句中的变量全部都是局部变量。与其他的语言不同，在 Prolog 中没有全局变量，取而代之的是 Prolog 的数据库。它使得所

有的 Prolog 子句能够共享信息。而 asserts 和 retracts 就是控制这些全局数据的工具。

使用全局数据有助于在子句之间快速的传递信息。不过，这种方式隐藏了子句之间的调用关系，所以一旦程序出错，是很难找到原因的。

我们完全也可以不使用 assert 和 retract 来完成上述的功能，不过这就需要把信息作为参数在子句中传递。在这种情况下，游戏中的状态将使用谓词的参数来储存，而不是谓词的子句。每一个谓词的入口参数是当前状态，而出口参数则为此谓词修改后的状态，状态在谓词之间传递，从而达到了预期的目的。我们还将在以后的章节中介绍这种方法。

我们现在所编写的程序并不都是从纯逻辑的考虑出发的，不过你可以看出使用 Prolog 编写这个游戏的过程非常自然，并没有什么晦涩难懂的东西。

一般情况下，asserta 等谓词是不会在回溯的时候还原数据库的，所以上面的几个数据管理谓词的内部流程与 I/O 谓词相同，不过我们可以很容易的编写出能够在回溯时取消修改的谓词。

```
backtracking_assert(X):- asserta(X).

backtracking_assert(X):- retract(X),fail.
```

首先第一个子句被运行，在数据库中添加一条 X 子句。当其后的目标失败而产生回溯时，第二个子句将被调用，于是它把第一个子句的操作给取消了，又把子句 X 从数据库中上除了。

第 10 章-递归

prolog-使用逻辑编程语言的教程，第十章 - 递归。主要包括 Prolog 下使用递归的方法、工作原理和优化等。

递归的确是一种功能强大的编程算法，现在绝大部分的程序语言都支持函数的递归调用，Prolog 也不例外，而且如果没有递归，Prolog 就不能叫做 Prolog 了。

在 Prolog 中，当某个谓词的目标中包含了此谓词本身时，Prolog 将进行递归调用。

正如前面所述的，某一规则被调用时，Prolog 使用新的变量为此规则的 body 部分建立新的查询。由于每次的查询都是独立的，所以某一规则调用其自身与调用其他规则没有任何区别。

任何语言中的递归定义都包括两个部分：边界条件与递归部分。

边界条件定义最简单的情况。而递归部分，则首先解决一部分问题，然后再调用其自身来解决剩下的部分，每一次都将进行边界检测，如果剩下的部分已经是边界条件中所定义的情况时，那么递归就圆满成功了。

下面我们将定义一个能够检测某物体在其他物体中的谓词，这里将使用到递归。

以前所定义的 location/2 谓词，表述了手电筒（flashlight）在桌子（desk）里，而桌子在办公室（office）中。但是那时 Prolog 并不能判断手电筒是否在办公室中。

```
?- location(flashlight, office).

no
```

如果使用递归，我们就可以很轻松地写出谓词 is_contained_in/2，它能够跟踪物体的所在的位置，因此它能判断手电筒是否在办公室中。

为了让问题更加有趣一些，我们再加入一些物品，它们的位置是一层一层地嵌套的。

```
location(envelope, desk).

location(stamp, envelope).

location(key, envelope).
```

要想列出办公室中的所有物品，我们首先可以列出直接位于办公室的物品，例如桌子；然后，再列出桌子中的物品，再桌子中的物品中的物品.....

如果把房间也看作一个物品的话，我们就可以很容易地写出具有两个部分的规则，它能够判断某物品是否在另一个物品中的。

如果物品 T1 直接位于物品 T2 中，则物品 T1 在物品 T2 中。(此为边界条件)

如果某一物品 X 直接位于 T2 中，而物品 T1 在物品 X 中（此处为递归调用），则物品 T1 在物品 T2 中。

用 Prolog 的语言来表达，上面的第一句可以写成，

```
is_contained_in(T1,T2) :- location(T1,T2).
```

而第二句则是，

```
is_contained_in(T1,T2) :- location(X,T2), is_contained_in(T1,X).
```

上面的递归很直接，请注意它是如何调用其自身的。

下面是此谓词的运行实例，

```
?- is_contained_in(X, office).
X = desk ;
X = computer ;
X = flashlight ;
X = envelope ;
X = stamp ;
X = key ;
no

?- is_contained_in(envelope, office).
yes

?- is_contained_in(apple, office).
no
```

递归的工作原理

规则中所定义的变量都是局部的。这意味着每次调用某一规则时，Prolog 都将为此次调用新建一个独立的变量集。因此递归第一层的变量 X、T1、T2，与第二层的变量 X、T1、T2 的变量名虽然相同，但是它们的值却是不同的。

我们可以使用带标号的变量或者 Prolog 的内部变量来区分这些局部变量。一开始，查询的目标是，

```
?- is_contained_in(XQ, office).
```

第一层递归的子句是：（在此使用带标号的变量来区分不同的递归级别，此处，T11 表示是 T1 在第一层递归中的变量）

```
is_contained_in(T11, T21) :- location(X1, T21), is_contained_in(T11, X1).
```

当查询的目标与此子句匹配时，变量的绑定情况如下：

```
XQ = _01

T11 = _01

T21 = office

X1 = _02
```

注意，查询目标中的变量 XQ 与 T11 同时绑定为_01，因此，一旦_01 的值找到了，则 XQ 和 T11 的值也就同时找到了。

使用这些绑定后的变量，可以重写上面的子句，

```
is_contained_in(_01, office) :-

location(_02, office), is_contained_in(_01, _02).
```

当 locatio/2 目标成功后，变量_02 绑定为 desk，即_02=desk，那么后面的递归调用就变成了，

```
is_contained_in(_01, desk)
```

这个新的目标将与 is_contained_in/2 的子句匹配，此时 Prolog 为本次匹配重新分配变量，，此时所有产生的变量如下，

```
XQ = _01 T11 = _01 T12 = _01

T21 = office T22 = desk

X1 = desk X2 = _03
```

当最后的递归找了某个答案，例如 envelope，则变量 T12、T11、XQ 将同时取为此值。下面是这个查询的详细步骤。

我们询问是，

```
?- is_contained_in(X, office).
```

递归的每一层都有自己独立的变量，但是正如调用其他的规则一样，上一层的变量会与正在调用的那一层的变量之间通过绑定联系起来，在下面的程序跟踪中，将使用 Prolog 的内部变量来说明，这样可以很容易知道那些变量被绑定了。

```
1-1 CALL is_contained_in(_0, office)

1-1 try (1) is_contained_in(_0, office)

2-1 CALL location(_0, office)

2-1 EXIT location(desk, office)

1-1 EXIT is_contained_in(desk, office)

X = desk ;

2-1 REDO location(_0, office)

2-1 EXIT location(computer, office)

1-1 EXIT is_contained_in(computer, office)

X = computer ;

2-1 REDO location(_0,office)

2-1 FAIL location(_0,office)
```

当没有更多的 location(X,office)子句时，is_contained_in/2 的第一条子句就失败了，Prolog 将试图满足第二条子句。请注意，在下面的调用中，location 子句没有使用以前的变量，而是一个新的内部变量，_4。而 T1 仍然与_0 绑定。

```
1-1 REDO is_contained_in(_0, office)

1-1 try (2) is_contained_in(_0, office)

2-1 CALL location(_4, office)

2-1 EXIT location(desk, office)
```

当对 is_contained_in/2 进行一次新的调用时，与我们在解释器的提示符后面直接输入 is_contained_in(X,desk)是完全相同的。这次调用将找出所有直接位于 desk 中的物品，正如上面找出直接位于 office 中的物品一样。

```
2-2 CALL is_contained_in(_0, desk)
```

2-2 try (1) is_contained_in(_0, desk)

3-1 CALL location(_0, desk)

3-1 EXIT location(flashlight, desk)

在第二层的 is_contained_in/2 中找到了 flashlight，这个答案将被传递到最上层的 is_contained_in/2 中。

2-2 EXIT is_contained_in(flashlight, desk)

1-1 EXIT is_contained_in(flashlight, office)

X = flashlight ;

同样，在第二层的递归中还找到了 envelope。

3-1 REDO location(_0, desk)

3-1 EXIT location(envelope, desk)

2-2 EXIT is_contained_in(envelope, desk)

1-1 EXIT is_contained_in(envelope, office)

X = envelope ;

找完了桌子里面的东西后，它开始找桌子里的东西的里面的东西。

3-1 REDO location(_0, desk)

3-1 FAIL location(_0, desk)

2-2 REDO is_contained_in(_0, desk)

2-2 try (2) is_contained_in(_0, desk)

3-1 CALL location(_7, desk)

3-1 EXIT location(flashlight, desk)

首先，看看 flashlight 里面还有没有东西，两个 is_contained_in/2 都失败了，因为在 flashlight 中找不到别的东西了。

3-2 CALL is_contained_in(_0, flashlight)

4-1 CALL location(_0, flashlight)

4-1 FAIL location(_0, flashlight)

3-2 REDO is_contained_in(_0, flashlight)

3-2 try (2) is_contained_in(_0, flashlight)

4-1 CALL location(_11, flashlight)

4-1 FAIL location(_11, flashlight)

3-2 FAIL is_contained_in(_0, flashlight)

下面，再开始找 envelope 中的 stamp。

3-1 REDO location(_7, desk)

3-1 EXIT location(envelope, desk)

3-2 CALL is_contained_in(_0, envelope)

4-1 CALL location(_0, envelope)

4-1 EXIT location(stamp, envelope)

3-2 EXIT is_contained_in(stamp, envelope)

2-2 EXIT is_contained_in(stamp, desk)

1-1 EXIT is_contained_in(stamp, office)

X = stamp ;

然后是 key。

4-1 REDO location(_0,envelope)

4-1 EXIT location(key, envelope)

3-2 EXIT is_contained_in(key, envelope)

2-2 EXIT is_contained_in(key, desk)

1-1 EXIT is_contained_in(key, office)

X = key ;

再没有别的东西的，于是就一路失败回去。

3-2 REDO is_contained_in(_0, envelope)

3-2 try (2) is_contained_in(_0, envelope)

4-1 CALL location(_11, envelope)

4-1 EXIT location(stamp, envelope)

4-2 CALL is_contained_in(_0, stamp)

5-1 CALL location(_0, stamp)

5-1 FAIL location(_0, stamp)

4-2 REDO is_contained_in(_0, stamp)

4-2 try(2) is_contained_in(_0, stamp)

5-1 CALL location(_14, stamp)

5-1 FAIL location(_14, stamp)

4-1 REDO location(_11, envelope)

4-1 EXIT location(key, envelope)

4-2 CALL is_contained_in(_0, key)

4-2 try (1) is_contained_in(_0, key)

5-1 CALL location(_0, key)

```
5-1 FAIL location(_0, key)

4-2 REDO is_contained_in(_0, key)

4-2 try (2) is_contained_in(_0, key)

5-1 CALL location(_14, key)

5-1 FAIL location(_14, key)

4-1 REDO location(_7, desk)

4-1 FAIL location(_7, desk)

3-1 REDO location(_4, office)

3-1 EXIT location(computer, office)

3-2 CALL is_contained_in(_0, computer)

4-1 CALL location(_0, computer)

4-1 FAIL location(_0, computer)

3-2 REDO is_contained_in(_0, computer)

4-1 CALL location(_7, computer)

4-1 FAIL location(_7, computer)

3-1 REDO location(_4, office)

3-1 FAIL location(_4, office)

no
```

优化

现在我们已经接触到了 Prolog 程序的一些神奇的地方，它所提供的编程方式不需要考虑程序的运行流程，而是注重于逻辑关系。但是，某些情况下，为了能够是程序快速的运行，我们不得不考虑这个问题。下面是一个例子。

首先目标 location(X,Y)将于任何 location/2 子句匹配，而目标 location(X,office)或 location(envelope,X)只能与某些子句匹配。

下面我们来看一看 is_contained_in/2 谓词的第二条子句的两种写法。

```
is_contained_in(T1,T2):- location(X,T2), is_contained_in(T1,X).
```

```
is_contained_in(T1,T2):- location(T1,X), is_contained_in(X,T2).
```

它们都可以找到正确的答案，但是它们的运行性能将取决于我们的询问方式。当询问是 is_contained_in(X,office)时，前者的运行速度较快。这是因为当 T2 绑定时，搜寻 location(X,T2)目标将比两个变量都不绑定时容易。同样，后者却能够较快地完成查询 is_contained_in(key,X)。

看样子，要想编出性能优越的程序还是要下一番工夫的啊。

第 11 章-联合

prolog-使用逻辑编程语言的教程，第 11 章-联合。主要包括 Prolog 的最强大的功能-它内建了模式匹配的算法-联合(Unification)。

Prolog 的最强大的功能之一就是它内建了模式匹配的算法----联合(Unification)。以前我们所介绍的例子中的联合都是较为简单的。现在来仔细研究一下联合。下表中列出了联合操作的简要情况。

变量&任何项目：变量可以与任何项目绑定，其中也包括变量

原始项目&原始项目：两个原始项目（原子或整数）只有当它们相同时才能联合。

结构&结构：如果两个结构的每个相应的参数能联合，那么这两个结构可以联合。

为了更清楚地介绍联合操作，我们将使用 Prolog 的内部谓词‘=/2’，此谓词当它的两个参数能够联合时成功，反之则失败。它的语法如下：

```
=(arg1, arg2)
```

为了方便阅读，也可以写成如下形式：

```
arg1 = arg2
```

注意：此处的等号在 Prolog 中的意义与其他语言中的不同。它不是数学运算符或者赋值符。

使用=进行联合操作与 Prolog 使用目标与子句联合时相同。在回溯时，变量将被释放。

下面举了几个最简单的联合的例子。

```
?- a = a.

yes


?- a = b.

no


?- location(apple, kitchen) = location(apple, kitchen).

yes


?- location(apple, kitchen) = location(pear, kitchen).

no


?- a(b,c(d,e(f,g))) = a(b,c(d,e(f,g))).

yes


?- a(b,c(d,e(f,g))) = a(b,c(d,e(g,f))).

no
```

在下面的例子中使用的变量，注意变量是如何绑定为某个值的。

```
?- X = a.

X = a


?- 4 = Y.
```

```
Y = 4


?- location(apple, kitchen) = location(apple, X).

X = kitchen
```

当然也可以同时使用多个变量。

```
?- location(X,Y) = location(apple, kitchen).

X = apple

Y = kitchen


?- location(apple, X) = location(Y, kitchen).

X = kitchen

Y = apple
```

变量之间也可以联合。每个变量都对应一个 Prolog 的内部值。当两个变量之间进行联合时，Prolog 就把它们标记为相同的值。在下面的例子中，我们假设 Prolog 使用‘_nn’，其中‘n’为数字，代表没有绑定的变量。

```
?- X = Y.

X = _01

Y = _01


?- location(X, kitchen) = location(Y, kitchen).

X = _01

Y = _01
```

Prolog 记住了被绑定在一起的变量，这将在后面的绑定中反映出来，请看下面的例子。

```
?- X = Y, Y = hello.

X = hello

Y = hello



?- X = Y, a(Z) = a(Y), X = hello.

X = hello

Y = hello

Z = hello
```

最后的这个例子能够很好地说明 Prolog 的变量绑定与其他语言中的变量赋值的区别。请仔细分析下面的询问。

```
?- X = Y, Y = 3, write(X).

3

X = 3

Y = 3



?- X = Y, tastes_yucky(X), write(Y).

broccoli

X = broccoli

Y = broccoli
```

当两个含变量的结构之间进行联合时，变量所取的值使得这两个结构相同。

```
?- X = a(b,c).

X = a(b,c)



?- a(b,X) = a(b,c(d,e)).

X = c(d,e)



?- a(b,X) = a(b,c(Y,e)).

X = c(_01,e)

Y = _01
```

无论多么复杂，Prolog 都将准确地记录下变量之间的关系，一旦某个变量绑定为某值，与之有关的变量都将改变。

```
?- a(b,X) = a(b,c(Y,e)), Y = hello.

X = c(hello, e)

Y = hello



?- food(X,Y) = Z, write(Z), nl, tastes_yucky(X), edible(Y), write(Z).
food(_01,_02)

food(broccoli, apple)

X = broccoli

Y = apple

Z = food(broccoli, apple)
```

如果在两次绑定中变量的值发生冲突，那么目标就失败了。

```
?- a(b,X) = a(b,c(Y,e)), X = hello.

no
```

上面的例子中,第二个子目标失败了,因为找不到一个 y 的值使得 hello 与 c(Y,e) 之间能够联合。而下面的例子是成功的。

```
?- a(b,X) = a(b,c(Y,e)), X = c(hello, e).

X = c(hello, e)

Y = hello
```

如果变量不能绑定为某一可能的值，那么联合也将失败。

```
?- a(X) = a(b,c).

no



?- a(b,c,d) = a(X,X,d).

no
```

下面的这个例子很有趣，请你研究一下吧。

```
?- a(c,X,X) = a(Y,Y,b).

no
```

你明白为什么这个例子失败么？第一个参数的绑定使得 Y 绑定为 c，第二个参数之间的绑定告诉 Prolog 变量 X 与 Y 的值相同，那么 X 也绑定 c，而最后一个参数的绑定使得 X 为 b，有矛盾，所以失败了。这就是说没有什么办法能使得这两个结构联合。

匿名变量（_）不会绑定为任何值。所以也不要求它所出现的位置的值必须相同。

```
?- a(c,X,X) = a(_,_,b).

X = b
```

如果使用（=）那么联合操作时显式的。而 Prolog 在使用子句与目标匹配时的联合则是隐式的。

第 12 章-数据结构

prolog-使用逻辑编程语言的教程，第十二章 - 数据结构。主要包括如何把最简单的数据组合起来，生成复杂的数据类型-结构。

到目前为止，所介绍的事实、查询以及规则都使用的是最简单的数据结构。谓词的参数都是原子或者整数，这些都是 Prolog 的基本组成元素。例如我们所使用过的原子有:

```
office, apple flashlight, nani
```

通过把这些最简单的数据组合起来,可以生成复杂的数据类型,我们称之为结构。结构由结构名和一定数量的参数组成。这与以前所学过的目标和事实是一样的。

```
functor(arg1,arg2,...)
```

结构的参数可以是简单的数据类型或者是另一个结构。现在在游戏中的物品都是由原子表示的，例如，desk、apple。但是使用结构可以更好的表达这些东西。下面的结构描述了物品的颜色、大小以及重量。

```
object(candle, red, small, 1).

object(apple, red, small, 1).

object(apple, green, small, 1).

object(table, blue, big, 50).
```

这些结构可以直接取代原来的 location/2 中的参数。但是这里我们再定义一个谓词 location_s/2。注意，虽然定义的结构较为复杂，但是它仍然是 location_s/2 的一个参数。

```
location_s(object(candle, red, small, 1), kitchen).

location_s(object(apple, red, small, 1), kitchen).

location_s(object(apple, green, small, 1), kitchen).

location_s(object(table, blue, big, 50), kitchen).
```

Prolog 的变量是没有数据类型之分的，所以它可以很容易的绑定为结构，如同它绑定为原子一样。事实上，原子就是没有参数的最简单的结构。因此可以有如下的询问。

```
?- location_s(X, kitchen).

X = object(candle, red, small, 1) ;

X = object(apple, red, small, 1) ;

X = object(apple, green, small, 1) ;

X = object(table, blue, big, 50) ;

no
```

我们还可以让变量绑定为结构中的某些参数，下面的询问可以找出厨房中所有红色的东西。

```
?- location_s(object(X, red, S, W), kitchen).

X = candle

S = small

W = 1 ;



X = apple

S = small
```

```
W = 1 ;



no
```

如果不关心大小和重量，可以使用下面的询问，其中变量‘_’是匿名变量。

```
?- location_s(object(X, red, _, _), kitchen).

X = candle ;

X = apple ;

no
```

使用这些结构，可以使得游戏更加真实。例如，我们可以修改以前所编写的 can_take/1 谓词，使得只有较轻的物品才能被玩家携带。

```
can_take_s(Thing) :-

here(Room),

location_s(object(Thing, _, small,_), Room).
```

同时，也可以把不能拿取某物品的原因说得更详细一些，现在有两个拿不了物品的原因。为了让 Prolog 在回溯时不把两个原因同时显示出来，我们为每个原因建立一条子句。这里要用到内部谓词 not/1，它的参数是一个目标，如果此目标失败，则它成功；目标成功则它失败。例如，

```
?- not( room(office) ).

no



?- not( location(cabbage, 'living room') )

yes
```

注意，在 Prolog 中的 not 的意思是：不能通过当前数据库中的事实和规则推出查询的目标。下面是使用 not 重新编写的 can_take_s/1。

```
can_take_s(Thing) :-

here(Room),

location_s(object(Thing, _, small, _), Room).

can_take_s(Thing) :-

here(Room),

location_s(object(Thing, _, big, _), Room),

write('The '), write(Thing),

write(' is too big to carry.'), nl,

fail.

can_take_s(Thing) :-

here(Room),

not (location_s(object(Thing, _, _, _), Room)),

write('There is no '), write(Thing), write(' here.'), nl,

fail.
```

下面来试试功能，假设玩家在厨房里。

```
?- can_take_s(candle).

yes



?- can_take_s(table).

The table is too big to carry.

no
```

```
?- can_take_s(desk).

There is no desk here.

no
```

原来的 list_things/1 谓词也可以加上一些功能，下面的 list_things_s/1 不但可以列出房间中的物品，还可以给出它们的描述。

```
list_things_s(Place) :-

location_s(object(Thing, Color, Size, Weight),Place),

write('A '),write(Size),tab(1),

write(Color),tab(1),

write(Thing), write(', weighing '),

write(Weight), write(' pounds'), nl,

fail.

list_things_s(_)
```

它的回答令人满意多了。

```
?- list_things_s(kitchen).

A small red candle, weighing 1 pounds

A small red apple, weighing 1 pounds

A small green apple, weighing 1 pounds

A big blue table, weighing 50 pounds
```

yes

如果你觉得使用 1 pounds 不太准确的话，我们可以再使用另一个谓词来解决此问题。

```
write_weight(1) :- write('1 pound').

write_weight(W) :- W > 1, write(W), write(' pounds').
```

下面试试看

```
?- write_weight(4).

4 pounds

yes



?- write_weight(1).

1 pound

yes
```

第一个子句中不需要使用 W=1 这样的判断，我们可以直接把 1 写到谓词的参数中，因为只有为 1 时是使用单数，其他情况下都使用复数。第二个子句中需要加入 W>1，要不然当重量为 1 时两条子句就同时满足。

结构可以任意的嵌套，下面使用 dimension 结构来描述物体的长、宽、高。

```
object(desk, brown, dimension(6,3,3), 90).
```

当然，也可以这样来表达物品的特性

object(desk, color(brown), size(large), weight(90))

下面是针对它的一条查询。

location_s(object(X, _, size(large), _), office).

要注意变量的位置哟，不要搞混了。

第 13 章-列表

prolog-使用逻辑编程语言的教程，第十三章 - 列表。主要包括如何在 Prolog 使用列表这种数据结构等。

为了能够更好地表达一组数据，Prolog 引入了列表(List)这种数据结构。 列表是一组项目的集合，此项目可以是 Prolog 的任何数据类型，包括结构和列表。列表的元素由方括号括起来，项目中间使用逗号分割。例如下面的列表列出了厨房中的物品。

```
[apple, broccoli, refrigerator]
```

我们可以使用列表来代替以前的多个子句。例如：

```
loc_list([apple, broccoli, crackers], kitchen).

loc_list([desk, computer], office).

loc_list([flashlight, envelope], desk).

loc_list([stamp, key], envelope).

loc_list(['washing machine'], cellar).

loc_list([nani], 'washing machine').
```

可见使用列表能够简化程序。

当某个列表中没有项目时我们称之为空表，使用“[]”表示。也可以使用 nil 来表示。下面的句子表示 hall 中没有东西。

```
loc_list([], hall)
```

变量也可以与列表联合,就像它与其他的数据结构联合一样。假如数据库中有了上面的子句，就可以进行如下的询问。

```
?- loc_list(X, kitchen).

X = [apple, broccoli, crackers]



?- [_,X,_] = [apples, broccoli, crackers].

X = broccoli
```

最后这个例子可以取出列表中任何一个项目的值，但是这种方法是不切实际的。你必须知道列表的长度，但是在很多情况下，列表的长度是变化的。

为了更加有效的使用列表，必须找到存取、添加和删除列表项目的方法。并且，我们应该不用对列表项目数和它们的顺序操心。

Prolog 提供的两个特性可以方便的完成以上任务。首先，Prolog 提供了把表头项目以及除去表头项目后剩下的列表分离的方法。其次，Prolog 强大的递归功能可以方便地访问除去表头项目后的列表。

使用这两个性质，我们可以编出一些列表的实用谓词。例如 member/2，它能够找到列表中的元素；append/3 可以把两个列表连接起来。这些谓词都是首先对列表头进行处理，然后使用递归处理剩下的列表。

首先，请看一般的列表形式。

[X | Y]

使用此列表可以与任意的列表匹配，匹配成功后，X 绑定为列表的第一个项目的值，我们称之为表头(head)。而 Y 则绑定为剩下的列表，我们称之为表尾(tail)。

下面我们看几个例子。

```
?- [a|[b,c,d]] = [a,b,c,d].

yes
```

上面的联合之所以成功，是因为等号两边的列表是等价的。注意表尾 tail 一定是列表，而表头则是一个项目，可以是表，也可以是其他的任何数据结构。下面的匹配失败，在“|”之后只能是一个列表，而不能是多个项目。

```
?- [a|b,c,d] = [a,b,c,d].

no
```

下面是其它的一些列表的例子。

```
?- [H|T] = [apple, broccoli, refrigerator].

H = apple

T = [broccoli, refrigerator]



?- [H|T] = [a, b, c, d, e].

H = a

T = [b, c, d, e]
```

```
?- [H|T] = [apples, bananas].

H = apples

T = [bananas]


?- [H|T] = [a, [b,c,d]]. 这个例子中的第一层列表有两个项目。

H = a

T = [[b, c, d]]


?- [H|T] = [apples]. 列表中只有一个项目的情况

H = apples

T = []


空表不能与[H|T]匹配，因为它没有表头。


?- [H|T] = [].

no
```

注意：最后这个匹配失败非常重要，在递归过程中经常使用它作为边界检测。即只要表不为空，那么它就能与[X|Y]匹配，当表为空时，就不能匹配，表示已经到达的边界条件。

我们还可以在第二个项目后面使用“|”，事实上，|前面的都是项目，后面的是一个表。

```
?- [One, Two | T] = [apple, sprouts, fridge, milk].
```

```
One = apple

Two = sprouts

T = [fridge, milk]
```

请注意下面的例子中变量是如何与结构绑定的。内部变量现实除了变量之间的联系。

```
?- [X,Y|T] = [a|Z].

X = a

Y = _01

T = _03

Z = [_01 | _03]
```

这个例子中，右边列表中的 Z 代表其表尾，与左边列表中的[Y|T]绑定。

```
?- [H|T] = [apple, Z].

H = apple

T = [_01]

Z = _01
```

上面的例子中，左边的表为 T 绑定为右边的表尾[Z]。

请仔细研究最后的这两个例子，表的联合对编制列表谓词是很有帮助的。

表可以看作是表头项目与表尾列表组合而成。而表尾列表又是由同样的方式组成的。所以表的定义本质上是递归定义。我们来看看下面的例子。

```
?- [a|[b|[c|[d|[]]]]] = [a,b,c,d].

yes
```

前面我们说过，列表是一种特殊的结构。最后的这个例子让我们对表的理解加深了。它事实上是一个有两个参数的谓词。第一个参数是表头项目，第二个参数是表尾列表。如果我们把这个谓词叫做 dot/2 的话，那么列表[a，b，c，d]可以表示为：

```
dot(a,dot(b,dot(c,dot(d,[]))))
```

事实上，这个谓词是存在的，至少在概念上是这样，我们用“.”来表示这个谓词，读作 dot。

我们可以使用内部谓词 display/1 来显示 dot，它和谓词 write/1 大致上相同，但是当它的参数为列表时将使用 dot 语法来显示列表。

```
?- X = [a,b,c,d], write(X), nl, display(X), nl.

[a,b,c,d]

.(a,.(b,.(c,.d(,[]))))



?- X = [Head|Tail], write(X), nl, display(X), nl.

[_01, _02]

.(_01,_02)



?- X = [a,b,[c,d],e], write(X), nl, display(X), nl.
```

```
[a,b,[c,d],e]

.(a,.(b,.(.(c,.(d,[])),.(e,[]))))
```

从这个例子中我们可以看出为什么不使用结构的语法来表示列表。因为它太复杂了，不过实际上列表就是一种嵌套式的结构。这一点在我们编制列表的谓词时应该牢牢地记住。

我们可以很容易地写出递归的谓词来处理列表。首先我们来编写谓词 member/2，它能够判断某个项目是否在列表中。

首先我们考虑边界条件，即最简单的情况。某项目是列表中的元素，如果此项目是列表的表头。写成 Prolog 语言就是：

```
member(H,[H|T]).
```

从这个子句我们可以看出含有变量的事实可以当作规则使用。

第二个子句用到了递归，其意义是：如果项目是某表的表尾 tail 的元素，那么它也是此列表的元素。

```
member(X,[H|T]) :- member(X,T).
```

完整的谓词如下：

```
member(H,[H|T]).

member(X,[H|T]) :- member(X,T).
```

请注意两个 member/2 谓词的第二个参数都是列表。由于第二个子句中的 T 也是一个列表，所以可以递归地进行下去。

```
?- member(apple, [apple, broccoli, crackers]).

yes


?- member(broccoli, [apple, broccoli, crackers]).

yes


?- member(banana, [apple, broccoli, crackers]).

no
```

下面是 member/2 谓词的单步运行结果。

我们的询问是

```
?- member(b, [a,b,c]).
```

1-1 CALL member(b,[a,b,c])

目标模板与第一个子句不匹配，因为 b 不是[a，b，c]列表的头部。但是它可以与第二个子句匹配。

1-1 try (2) member(b,[a,b,c])

第二个子句递归调用 member/2 谓词。

2-1 CALL member(b,[b,c])

这时，能够与第一个子句匹配了。

2-1 EXIT (1) member(b,[b,c])

于是一直成功地返回到我们的询问子句。

1-1 EXIT (2) member(b,[a,b,c])

yes

和大部分 Prolog 的谓词一样，member/2 有多种使用方法。如果询问的第一参数是变量，member/2 可以把列表中所有的项目找出来。

?- member(X, [apple, broccoli, crackers]).

X = apple ;

X = broccoli ;

X = crackers ;

no

下面我们将使用内部变量来跟踪 member/2 的这种使用方法。请记住每一层递归都会产生自己的变量，但是它们之间通过模板联合在一起。

由于第一个参数是变量，所以询问的模板能够与第一个子句匹配，并且变量 X 将绑定为表头。回显出 X 的值后，用户使用分号引起回溯，Prolog 继续寻找更多的答案，与第二个子句进行匹配，这样就形成了递归调用。

我们的询问是

?- member(X,[a,b,c]).

当 X=a 时，目标能够与第一个子句匹配。

1-1 CALL member(_0,[a,b,c])

1-1 EXIT (1) member(a,[a,b,c])

X = a ;

回溯时释放变量，并且开始考虑第二条子句。

1-1 REDO member(_0,[a,b,c])

1-1 try (2) member(_0,[a,b,c])

第二层也成功了，和第一层相同。

2-1 CALL member(_0,[b,c])

2-1 EXIT (1) member(b,[b,c])

1-1 EXIT member(b,[a,b,c])

X = b ;

继续第三层，和前面相似。

2-1 REDO member(_0,[b,c])

2-1 try (2) member(_0,[b,c])

3-1 CALL member(_0,[c])

3-1 EXIT (1) member(c,[c])

```
2-1 EXIT (2) member(c,[b,c])

1-1 EXIT (2) member(c,[a,b,c])

X = c ;
```

下面试图找到空表中的元素。而空表不能与两个子句中的任何一个表匹配，所以查询失败了。

```
3-1 REDO member(_0,[c])

3-1 try (2) member(_0,[c])

4-1 CALL member(_0,[])

4-1 FAIL member(_0,[])

3-1 FAIL member(_0,[c])

2-1 FAIL member(_0,[b,c])

1-1 FAIL member(_0,[a,b,c])

no
```

下面再介绍一个有用的列表谓词。它能够把两个列表连接成一个列表。此谓词是append/3。第一个参数和第二个参数连接的表为第三个参数。例如：

```
?- append([a,b,c],[d,e,f],X).

X = [a,b,c,d,e,f]
```

这个地方有一个小小的麻烦，因为最基本的列表操作只能取得列表的头部，而不能在内表尾部添加项目。append/3 使用递归地减少第一个列表长度的方法来解决这个问题。

边界条件是：如果空表与某个表连接就是此表本身。

```
append([],X,X).
```

而递归的方法是：如果列表[H|T1]与列表 X 连接，那么新的表的表头为 H，表尾则是列表 T1 与 X 连接的表。

```
append([H|T1],X,[H|T2]) :- append(T1,X,T2)
```

完整的谓词就是：

```
append([],X,X).
append([H|T1],X,[H|T2]) :- append(T1,X,T2).
```

Prolog 真正独特的地方就在这里了。在每一层都将有新的变量被绑定，它们和上一层的变量联合起来。第二条子句的递归部分的第三个参数 T2，与其头部的第三个参数的表尾相同，这种关系在每一层中都是使用变量的绑定来体现的。下面是跟踪运行的结果。

我们的询问是：

```
  ?- append([a,b,c],[d,e,f],X).
1-1 CALL append([a,b,c],[d,e,f],_0)
X = _0
2-1 CALL append([b,c],[d,e,f],_5)
_0 = [a|_5]
```

```
3-1 CALL append([c],[d,e,f],_9)

_5 = [b|_9]

4-1 CALL append([],[d,e,f],_14)

_9 = [c|_14]
```

把变量的所有联系都考虑进来，我们可以看出，这时变量 X 有如下的绑定值。

```
X = [a|[b|[c|_14]]]
```

到达了边界条件，因为第一个参数已经递减为了空表。与第一条子句匹配时，变量_14 绑定为表[d，e，f]，这样我们就得到了 X 的值。

```
4-1 EXIT (1) append([],[d,e,f],[d,e,f])

3-1 EXIT (2) append([c],[d,e,f],[c,d,e,f])

2-1 EXIT (2) append([b,c],[d,e,f],[b,c,d,e,f])

1-1 EXIT (2)append([a,b,c],[d,e,f],[a,b,c,d,e,f])

X = [a,b,c,d,e,f]
```

和 member/2 一样，append/3 还有别的使用方法。下面这个例子显示了 append/3 是如何把一个表分解的。

```
?- append(X,Y,[a,b,c]).

X = []

Y = [a,b,c] ;



X = [a]
```

```
Y = [b,c] ;



X = [a,b]

Y = [c] ;



X = [a,b,c]

Y = [] ;

no
```

使用列表

现在有了能够处理列表的谓词，我们就可以在游戏中使用它们。例如使用谓词 loc_list/2 代替原来的谓词 location/2 来储存物品，然后再重新编写 location/2 来完成与以前同样的操作。只不过是以前是通过 location/2 寻找答案,而现在是使用 location/2 计算答案了。这个例子从某种程度上说明了 Prolog 的数据与过程之间没有明显的界限。无论是从数据库中直接找到答案，或是通过一定的计算得到答案，对于调用它的谓词来说都是一样的。

```
location(X,Y):- loc_list(List, Y), member(X, List).
```

当某个物品被放入房间时，需要修改此房间的 loc_lists，我们使用 append/3 来编写谓词 add_thing/3：

```
add_thing(NewThing, Container, NewList):-

loc_list(OldList, Container),

append([NewThing],OldList, NewList).
```

其中，NewThing 是要添加的物品，Container 是此物品的位置，NewList 是添加物品后的列表。

```
?- add_thing(plum, kitchen, X).

X = [plum, apple, broccoli, crackers]
```

当然，也可以直接使用[Head|Tail]这种列表结构来编写 add_thing/3。

```
add_thing2(NewThing, Container, NewList):-

loc_list(OldList, Container),

NewList = [NewThing | OldList].
```

它和前面的 add_thing/3 功能相同。

```
?- add_thing2(plum, kitchen, X).

X = [plum, apple, broccoli, crackers]
```

我们还可以对 add_thing2/3 进行简化，不是用显式的联合，而改为在子句头部的隐式联合。

```
add_thing3(NewTh, Container,[NewTh|OldList]) :-

loc_list(OldList, Container).
```

它同样能完成我们的任务。

```
?- add_thing3(plum, kitchen, X).

X = [plum, apple, broccoli, crackers]
```

下面的 put_thing/2，能够直接修改动态数据库，请自己研究一下。

```
put_thing(Thing,Place) :-

retract(loc_list(List, Place)),

asserta(loc_list([Thing|List],Place)).
```

到底是使用多条子句，还是使用列表方式，这完全有你的编程习惯来决定。有时使用 Prolog 的自动回溯功能较好，而有时则使用递归的方式较好。还有些较为复杂的情况，需要同时使用子句和列表来表达数据。 这就必须掌握两种数据表达方式之间的转换。

把一个列表转换为多条子句并不难。使用递归过程逐步地把表头 asserts 到数据库中就行了。下面的例子把列表转化为了 stuff 的一系列子句。

```
break_out([]).

break_out([Head | Tail]):-

assertz(stuff(Head)),

break_out(Tail).



?- break_out([pencil, cookie, snow]).

yes
```

```
?- stuff(X).

X = pencil ;

X = cookie ;

X = snow ;

no
```

把多条事实转化为列表就困难多了。因此 Prolog 提供了一些内部谓词来完成这个任务。最常用的谓词是 findall/3，它的参数意义如下：

参数 1： 结果列表的模板。

参数 2： 目标模板。

参数 3： 结果列表。

findall/3 自动地寻找目标，并把结果储存到一个列表中。使用它可以方便的把 stuff 子句还原成列表。

```
?- findall(X, stuff(X), L).

L = [pencil, cookie, snow]
```

下面把所有与厨房相连的房间找出来。

```
?- findall(X, connect(kitchen, X), L).

L = [office, cellar, 'dining room']
```

最后我们再来看一个复杂的例子：

```
?- findall(foodat(X,Y), (location(X,Y) , edible(X)), L).
L = [foodat(apple, kitchen), foodat(crackers, kitchen)]
```

它找出了所有能吃的东西及其位置，并把结果放到了列表中。

第 14 章－操作符

prolog-使用逻辑编程语言的教程，第十四章 - 操作符。主要包括 Prolog 中数学操作符和学习如何定义自己的操作符的语法等。

我们已经学习过了 Prolog 的数据结构,它的形式如下：

functor(arg1,arg2,...,argN).

这是 Prolog 的唯一的数据结构，但是 Prolog 允许这种数据结构有其它的表达方法（仅仅是表达方法不同）。这种表达方法有时候更加接近我们的习惯，正如列表的两种表达法一样。现在要介绍的是操作符语法。

以前曾经介绍了数学符号，在这一章我们将看到它和 Prolog 的数据结构是等价的，并且学习如何定义自己的操作符。

所有的数学操作符都是 Prolog 的基本符号，例如-/2、+/2、-/1。使用谓词 display/1 可以看到它们的标准的语法结构。

```
?- display(2 + 2).

+(2,2)



?- display(3 4 + 6).

+((3,4),6)



?- display(3 (4 + 6)).

(3,+(4,6))
```

你可以把任何谓词定义为操作符的形式，例如，如果我们把 location/2 定义为了操作符，那么我们就可以用：

apple location kitchen.

来代替

location(apple, kitchen).

注意：这只是书写形式上的不同，在 Prolog 进行模式匹配时它们都是一样的。

操作符有三种形式：

- 中缀（infix）：例如 3+4
- 前缀（prefix）：例如-7
- 后缀（postfix）：例如 8 factorial

每个操作符有不同的优先权值，从 1 到 1200。当某句中有多个操作符时，优先权高的将先被考虑。优先权值越小优先权越高。

使用内部谓词 op/3 来定义操作符，它的三个参数分别是：优先权、结合性、操作符名称。

结合性使用模板来定义，例如中缀操作符使用“xfx”来定义。“f”表示操作符的位置。

下面我们将重新编写 location/2 谓词，并改名为 is_in/2。

is_in(apple, room(kitchen)).

使用 op/3 谓词把 is_in/2 定义为操作符，优先权值为 35。

?- op(35,xfx,is_in).

下面是我们的询问。

```
?- apple is_in X.

X = room(kitchen)



?- X is_in room(kitchen).

X = apple
```

同样可以使用操作符来定义事实。

```
banana is_in room(kitchen).
```

为了证明这两种数据结构是等价，我们可以进行如下的比较：

```
?- is_in(banana, room(kitchen)) = banana is_in room(kitchen).

yes
```

使用 display/1 可以清楚地看到这一点。

```
?- display(banana is_in room(kitchen)).

is_in(banana, room(kitchen))
```

下面再把 room/1 定义为前缀操作符。前缀操作符的模板是 fx。它的优先权应该比 is_in 的高。这里取 33。

```
?- op(33,fx,room).



?- room kitchen = room(kitchen).

yes



?- apple is_in X.

X = room kitchen
```

使用上面的两个操作符，我们可以使用如下的方式定义事实。

```
pear is_in room kitchen.



?- is_in(pear, room(kitchen)) = pear is_in room kitchen.

yes



?- display(pear is_in room kitchen).

is_in(pear, room(kitchen))
```

注意如果操作符的优先权搞错了，那就全部乱了套。例如：如果 room/1 的优先权低于 is_in/2，那么上面的结构就变成了下面这个样子：

```
room(is_in(apple, kitchen))
```

不但如此，Prolog 的联合操作也将出现问题。所以一定要仔细考虑操作符的优先权。

最后我们来定义后缀操作符，使用模板 xf。

```
?- op(33,xf,turned_on).


flashlight turned_on.


?- turned_on(flashlight) = flashlight turned_on.

yes
```

使用操作符可以是程序更容易阅读。

在我们的命令驱动的“寻找 Nani”游戏中，为了使发出的命令更接近自然语言，可以使用操作符来定义。

```
goto(kitchen) -> goto kitchen.

turn_on(flashlight) -> turn_on flashlight.

take(apple) -> take apple.
```

虽然这还不是真正的自然语言，可是比起带括号的来还是方便多了。

当操作符的优先权相同时，Prolog 必须决定是从左到右还是从右到左地读入操作符。这就是操作符的左右结合性。有些操作符没有结合性，如果你把两个这种操作符放到一起将产生错误。

下面是结合性的模板：

```
Infix:

xfx non-associative （没有结合性）

xfy right to left

yfx left to right

Prefix

fx non-associative

fy left to right

Postfix:

xf non-associative

yf right to left
```

前面所定义的谓词 is_in/2 没有结合性，所以下面的句子是错误的。

```
key is_in desk is_in office.
```

为了表示这种嵌套关系，我们可以使用从右到左的结合性。

```
?- op(35,xfy,is_in).

yes


?- display(key is_in desk is_in office).

is_in(key, is_in(desk, office))
```

如果使用从左到右的结合性，我们的结果将不同。

```
?- op(35,yfx,is_in).

yes

?- display(key is_in desk is_in office).

is_in(is_in(key, desk), office)
```

但是使用括号可以改变这种结合性：

```
?- display(key is_in (desk is_in office)).

is_in(key, is_in(desk, office))
```

由许多内部谓词都定义为了中缀操作符。因此我们可以使用“arg1 predicate arg2. ”来代替 predicate(arg1,arg2) 。

我们所见过的数学符号就是如此，例如+-/。但是一定要牢记这只是表达形式上的区别，因此 3+4 和 7 是不一样的，它就是+（3，4）。

只有一些特殊的内部谓词（例如 is/2）进行真正的数学运算。is/2 计算它右边表达式的值，并让左边绑定为此值。它与联合（=）谓词是不同的，=只进行联合而不进行计算。

```
?- X is 3 + 4.

X = 7



?- X = 3 + 4.

X = 3 + 4
```

```
?- 10 is 5 2.

yes


?- 10 = 5 2.

no


?- X is 3 4 + (6 / 2).

X = 15


?- X = 3 4 + (6 / 2).

X = 3 4 + (6 / 2)


?- X is +((3,4) , /(6,2)).

X = 15


?- 3 [4 + (6 / 2) = +(:  create](3,4),/(6,2)).

yes
```

只有当使用 is/2 来计算时，数学操作符才显示出其不同之处，而一般情况下与其它的谓词没有任何区别。

```
?- X = 3 4 + likes(john, 6/2).

X = 3 4 + likes(john, 6/2).


?- X is 3 4 + likes(john, 6/2).
```

```
error
```

我们已经知道 Prolog 的程序是由一系列的子句构成的。其实这些子句也是使用操作符书写的 Prolog 的数据结构。这里的操作符是":-"，它是中缀操作符，有两个参数。

```
:-(Head, Body).
```

Body 也是由操作符书写的数据结构。这里的操作符为","，它表示并且的意思，所以 Body 的形式如下：

```
,(goal1, ,(goal2,,goal3))
```

好像看不明白，操作符","与分隔符","无法区别，所以我们就是用"&"来代替操作符","，于是上面的形式就变成了下面这个样子了。

```
&(goal1, &(goal2, & goal3))
```

下面的两种形式表达的意思是相同的。

```
head :- goal1 & goal2 & goal3.

:-(head, &(goal1, &(goal2, & goal3))).
```

实际上是下面的形式：

```
head :- goal1 , goal2 , goal3.

:-(head, ,(goal1, ,(goal2, , goal3))).
```

数学操作符不但可以用来计算，还有许多其它的用途。例如 write/1，只能有一个参数，当我们想同时显示两个变量的值时， 就可以使用下面的方法。

```
?- X = one, Y = two, write(X-Y).

one - two
```

因为 X-Y 实际上是一个数据结构，所以它相对于 write 来说就只是一个参数。

当然其它的数学操作符也能完成相同的功能，例如/。在有些 Prolog 的版本中干脆引入了“：”这个操作符来专门完成这种任务，有了它我们可以很方便的书写复杂的数据结构了。

```
object(apple, size:small, color:red, weight:1).



?- object(X, size:small, color:C, weight:W).

X = apple

C = red

W = 1
```

这里我们使用 size:small，代替了原来的 size(small)，实际上“：”是中缀操作符，它的原始表达形式是:(size,small)。

从这一章所介绍的内容我们可以发现 Prolog 的程序实际上也是一种数据结构，只不过是使用专门的操作符连接起来的。那么到现在为止，我们所学习过的所有

Prolog 内容：事实、规则、结构、列表等的实质都是一样的，这也正是 Prolog 与其它语言的最大区别---程序与数据的高度统一。正是它的这种极其简洁的表达形式，使得它被广泛地应用于人工智能领域。

第 15 章-截断

prolog-使用逻辑编程语言的教程，第十五章 - 截断。主要包括 Prolog 中的截断-cut，如何使用 cut 等。

直到目前为止，我们都一直在使用 Prolog 内建的回溯功能。使用此功能可以方便地写出结构紧凑的谓词来。

但是，并不是所有的回溯都是必须的，这时我们需要能够人工地控制回溯过程。Prolog 提供了完成此功能的谓词，他叫做 cut，使用符号！来表示。

Cut 能够有效地剔除一些多余的搜索。如果在 cut 处产生回溯，它会自动地失败，而不去进行其它的选择。

下面我们将看看它的一些实际的功效。

请参照上图来理解 cut 的功能。当在回溯遇到 cut 时，它改变了回溯的流程，它直接把控制权传给了上一级目标，而不是它左边的目标。这样第一层的中间的那个目标以及第二层！左边的子目标都不会被 Prolog 重新满足。

下面我们将举个简单的例子来说明 cut 的作用。首先加入几条事实：

```
data(one).

data(two).

data(three).
```

下面是没有使用 cut 的情况：

```
cut_test_a(X) :- data(X).

cut_test_a('last clause').
```

下面是对上面的事实与规则的一次询问。

```
?- cut_test_a(X), write(X), nl, fail.

one

two

three

last clause

no
```

我们再来看看使用了 cut 之后的情况。

```
cut_test_b(X) :- data(X), !.

cut_test_b('last clause').



?- cut_test_b(X), write(X), nl, fail.

one

no
```

我们可以看到，由于在 cut_test_b(X)子句加入了 cut，data/1 子目标与 cut_test_b 父目标都没有产生回溯。

下面我们看看把 cut 放到两个子目标中的情况。

```
cut_test_c(X,Y) :- data(X), !, data(Y).

cut_test_c('last clause').



?- cut_test_c(X,Y), write(X-Y), nl, fail.

one - one

one - two

one - three

no
```

cut 抑制了其左边的子目标 data(X)与 cut_test_c 父目标的回溯，而它右边的目标则不受影响。

cut 是不符合纯逻辑学的，不过出于实用的考虑，它还是必须的。过多地使用 cut 将降低程序的易读性和易维护性。它就像是其它语言中的 goto 语句。

当你能够确信在谓词中的某一点只有一个答案，或者没有答案时，使用 cut 可以提高程序的效率，另外，如果在某种情况下你想让某个谓词强制失败，而不让它去寻找更多的答案时，使用 cut 也是个不错的选择。

下面将介绍使用 cut 的技巧。

使用 Cut

为了让冒险游戏更加有趣，我们来编写一个小小的迷题。我们把这个迷题叫做 puzzle/1。puzzle 的参数是游戏中的某个命令，puzzle 将判断这个命令有没有特殊的要求，并做出反应。

我们将在 puzzle/1 中见到 cut 的两种用法。我们想要完成的任务是：

如果存在 puzzle，并且约束条件成立，就成功。

如果存在 puzzle，而约束条件不成立，就失败。

如果没有 puzzle，成功。

在本游戏中的 puzzle 是要到地下室（cellar）中去，而玩家必须拥有手电筒，并且打开了，才能够进到地下室中。如果这些条件都满足了，就不需要 Prolog 再去进行其它的搜索。所以这里我们可以使用 cut。

```
puzzle(goto(cellar)):-

have(flashlight),

turned_on(flashlight),

!.
```

如果约束条件不满足，Prolog 就会通知玩家不能执行命令的原因。在这种情况下，我们也想 puzzle 谓词失败，而不去匹配其它的 puzzle 子句。因此，此处我们也使用 cut 来阻止回溯，并且在 cut 的后面加上 fail。

最后一个子句包括了所有非特殊的命令。这里我们看到，使用 cut 就像其它语言中的 if 语句一样，可以用它来判断不同的情况。

```
puzzle(_).
```

从纯逻辑的角度来看，能找到不使用 cut 而完成同样功能的方法。这时需要使用内部谓词 not/1。有人认为使用 not/1 可以使程序更加清晰，不过滥用 not 同样也会引起混乱的。

当使用 cut 时，子句的顺序显得尤为重要了。上例中，puzzle/1 的第二个子句可以直接打出错误信息，这是因为我们知道只有当第一个子句在遇到 cut 前失败时，Prolog 才会考虑第二个子句。

而第三个子句考虑的是最一般的情况，这是因为，前面两个子句已经考虑了特殊的情况。

如果把所有的 cut 都去掉，我们就必须改写第二、三个子句。

```
puzzle(goto(cellar)):-

not(have(flashlight)),

not(turned_on(flashlight)),

write('Scared of dark message'),

fail.



puzzle(X):-

not(X = goto(cellar)).
```

在这种情况下，子句的顺序就无关紧要了。有趣的是，事实上 not/1 子句可以使用 cut 来定义，它同时还用到了另一个内部谓词 call/1。call/1 把它的参数作为谓词来调用。

```
not(X) :- call(X), !, fail.

not(X).
```

在下一章中我们将学习如何在游戏中加入命令循环。那时我们就可以在每次运行玩家的命令之前使用 puzzle/1 来检验它。这里我们先试试 puzzle 的功能。

```
goto(Place) :-
```

```
puzzle(goto(Place)),

can_go(Place),

move(Place),

look.
```

如果玩家现在在厨房里，并且想到地下室中去。

```
?- goto(cellar).

It's dark and you are afraid of the dark.

no



?- goto(office).

You are in the office...
```

而如果玩家拿着打开的手电筒，它就可以去地下室了。

```
  ?- goto(cellar).

You are in the cellar...
```

第 16 章-流程控制

prolog-使用逻辑编程语言的教程，第十六章 - 流程控制。主要介绍和一般的程序设计语言相似的流程控制如循环等。

在前面的章节中，我们了解了 Prolog 是如何解释目标的，并且通过实例说明了 Prolog 的运行流程。

在这一章，继续探索 Prolog 的程序流程控制，我们将介绍和一般的程序设计语言相似的流程控制。

前面我们使用谓词 fail 和 write/1 来列印出游戏中所有的物品。这种流程控制类似于一般语言中“do，while”语句。

现在介绍另外一个使用失败来完成相同功能的内部谓词---repeat/0。它在第一次调用时永远成功，并且在回溯时也永远成功。换句话说，流程不可能回溯通过 repeat/0。

如果某个子句中有 repeat/0，并且其后有 fail/0 谓词出现，那么将永远循环下去。使用这种方法可以编写死循环的 Prolog 程序。

如果在 repeat/0 谓词后面加入几个中间目标，并且最后使用一个测试条件结束，那么程序将一直循环到条件满足为止。这相当于其它程序语言中的“do until”。在编写“寻找 Nani”这个游戏时，我们正好需要这种流程来编写最高层的命令循环。

我们先来看一个例子，它只是简单的读入命令并且在屏幕上回显出来，直到用户输入了 end 命令。内部谓词 read/1 可以从控制台对入一条 Prolog 字符串。此字符串必须使用“.”结束，就像所有的 Prolog 子句一样。

```
command_loop:-
```

```
repeat,

write('Enter command (end to exit): '),

read(X),

write(X), nl,

X = end.
```

最后面的那个目标 x=end 只有在用户输入 end 时才会成功，而 repeat/0 在回溯时将永远成功，所以这种结构将使得中将的目标能够被重复执行。

下面我们要做的事就是加入中间的命令执行部分，而不是简单的回显用户输入的命令。

我们先来编写一个新的谓词 do/1，它用来执行我们需要的谓词。在许多程序语言中，这种结构叫做“do case”，而在 Prolog 中我们使用多条子句来完成相同的功能。

下面是 do/1 的程序，我们可以使用 do/1 来定义命令的同义词，例如玩家可以输入 goto(X)或者 go(X)，这都将运行 goto(X)子句。

```
do(goto(X)):-goto(X),!.

do(go(X)):-goto(X),!.

do(inventory):-inventory,!.

do(look):-look,!.
```

此处的 cut 有两个用途。第一，如果我们找到了一条该执行的 do 子句，就没有必要去寻找更多的 do 子句了；其二，它有效地防止了在回溯时又重新运行 read 目标。

下面是另外的几条 do/1 的子句。如果没有 do(end)子句，那么条件 X=end 就永远不会成立，所以 end 是结束游戏的命令。最后一个 do/1 子句考虑不合法的命令。

```
do(take(X)) :- take(X), !.

do(end).

do(_) :-

write('Invalid command').
```

下面我们开始正式编写 command_loop/0 谓词，这里使用前面说编写的 puzzle/1 和本章介绍的 do/1 谓词来完成命令的解释工作。并且我们将考虑游戏结束的情况，游戏有两种结束方式，可以是玩家输入了 end 命令，或者玩家找到了 Nani。我们将编写一个新的谓词 end_condition/1 来完成这个任务。

```
command_loop:-

write('Welcome to Nani Search'), nl,

repeat,

write('>nani> '),

read(X),

puzzle(X),

do(X), nl,

end_condition(X).



end_condition(end).

end_condition(_) :-

have(nani),
```

```
write('Congratulations').
```

递归循环

在 Prolog 程序中使用 assert 和 retract 谓词动态地改变数据库的方法，不是纯逻辑程序的设计方法。就像其他语言中的全局变量一样，使用这种谓词会产生一些不可预测的问题。由于使用了这种谓词，可是会导致程序中两个本来应该独立的部分互相影响。

例如，puzzle(goto(cellar))的结果依赖于 turned_on(flashlight)是否存在于数据库中，而 turned_on(flashlight)是使用 turn_on 谓词动态地加入到数据库中的。所以如果 turn_on/1 中间有错误，它就会直接影响到 puzzle，这中程序之间的隐形联系正是造成错误的罪魁祸首。

我们可以重新改造程序，只使用参数传递信息，而不是全局数据。可以把这种情况想象成一系列的状态转换。

在本游戏中，游戏的状态是使用 location/2、here/1、have/1 以及 turned_on/1(turned_off/1)来定义的。我们首先使用这些谓词定义游戏的初始状态，其后玩家的操作将使用 assert 和 retract 动态地改变这些状态，直到最后达到了 have(nani)。

我们可以通过定义一个复杂的结构来储存游戏的状态来完成相同的功能，游戏的命令将把这个结构当作参数进行操作，而不是动态数据库。

由于逻辑变量是不能通过赋值来改变它们的值的，所以所有的命令都必须有两个参数，一个是旧的状态，另一个实行的状态。使用前面的 repeat-fail 循环结构无法完成参数的传递过程，因此我们就使用递归程序把状态传给它自己，而边界条件则是到达了游戏的最终状态。下面的程序就是使用这种方法编制而成的。

游戏的状态使用列表储存，列表的每个元素就是我们前面所定义的状态谓词，请看 initial_state/1 谓词。而每个命令都要对这个列表有不同的操作，谓词

get_state/3, add_state/4, 和 del_state/4 就是完成这个任务的，它们提供了操作状态列表的方法。

这种 Prolog 程序就是纯逻辑的，它完全避免的使用全局数据的麻烦。但是它需要更复杂的谓词来操作参数中的状态。而列表操作与递归程序则是最难调试的了。至于使用哪种方法就要有你决定了。

```
% a nonassertive version of nani search


nani :-

write('Welcome to Nani Search'),

nl,

initial_state(State),

control_loop(State).


control_loop(State) :-

end_condition(State).

control_loop(State) :-

repeat,

write('> '),

read(X),

constraint(State, X),

do(State, NewState, X),

control_loop(NewState).


% initial dynamic state
```

```
initial_state([
here(kitchen),
have([]),
location([
kitchen/apple,
kitchen/broccoli,
office/desk,
office/flashlight,
cellar/nani ]),
status([
flashlight/off,
game/on]) ]).

% static state

rooms([office, kitchen, cellar]).

doors([office/kitchen, cellar/kitchen]).

connect(X,Y) :-
doors(DoorList),
member(X/Y, DoorList).
```

```
connect(X,Y) :-
doors(DoorList),
member(Y/X, DoorList).

% list utilities

member(X,[X|Y]).
member(X,[Y|Z]) :- member(X,Z).

delete(X, [], []).
delete(X, [X|T], T).
delete(X, [H|T], [H|Z]) :- delete(X, T, Z).

% state manipulation utilities

get_state(State, here, X) :-
member(here(X), State).
get_state(State, have, X) :-
member(have(Haves), State),
member(X, Haves).
get_state(State, location, Loc/X) :-
member(location(Locs), State),
member(Loc/X, Locs).
```

```
get_state(State, status, Thing/Stat) :-
member(status(Stats), State),
member(Thing/Stat, Stats).

del_state(OldState, [location(NewLocs) | Temp], location, Loc/X):-
delete(location(Locs), OldState, Temp),
delete(Loc/X, Locs, NewLocs).

add_state(OldState, [here(X)|Temp], here, X) :-
delete(here(_), OldState, Temp).
add_state(OldState, [have([X|Haves])|Temp], have, X) :-
delete(have(Haves), OldState, Temp).
add_state(OldState, [status([Thing/Stat|TempStats])|Temp],
status, Thing/Stat) :-
delete(status(Stats), OldState, Temp),
delete(Thing/_, Stats, TempStats).

% end condition

end_condition(State) :-
get_state(State, have, nani),
write('You win').
end_condition(State) :-
```

```
get_state(State, status, game/off),
write('quitter').

% constraints and puzzles together

constraint(State, goto(cellar)) :-
!, can_go_cellar(State).
constraint(State, goto(X)) :-
!, can_go(State, X).
constraint(State, take(X)) :-
!, can_take(State, X).
constraint(State, turn_on(X)) :-
!, can_turn_on(State, X).
constraint(_, _).

can_go(State,X) :-
get_state(State, here, H),
connect(X,H).
can_go(_, X) :-
write('You can''t get there from here'),
nl, fail.

can_go_cellar(State) :-
```

```
can_go(State, cellar),
!, cellar_puzzle(State).

cellar_puzzle(State) :-
get_state(State, have, flashlight),
get_state(State, status, flashlight/on).
cellar_puzzle(_) :-
write('It''s dark in the cellar'),
nl, fail.

can_take(State, X) :-
get_state(State, here, H),
get_state(State, location, H/X).
can_take(State, X) :-
write('it is not here'),
nl, fail.

can_turn_on(State, X) :-
get_state(State, have, X).
can_turn_on(_, X) :-
write('You don''t have it'),
nl, fail.
```

```
% commands


do(Old, New, goto(X)) :- goto(Old, New, X), !.

do(Old, New, take(X)) :- take(Old, New, X), !.

do(Old, New, turn_on(X)) :- turn_on(Old, New, X), !.

do(State, State, look) :- look(State), !.

do(Old, New, quit) :- quit(Old, New).

do(State, State, _) :-

write('illegal command'), nl.


look(State) :-

get_state(State, here, H),

write('You are in '), write(H),

nl,

list_things(State, H), nl.


list_things(State, H) :-

get_state(State, location, H/X),

tab(2), write(X),

fail.

list_things(_, _).


goto(Old, New, X) :-
```

```
add_state(Old, New, here, X),

look(New).



take(Old, New, X) :-

get_state(Old, here, H),

del_state(Old, Temp, location, H/X),

add_state(Temp, New, have, X).



turn_on(Old, New, X) :-

add_state(Old, New, status, X/on).



quit(Old, New) :-

add_state(Old, New, status, game/off).
```

使用这种递归的方法来完成任务，还有一个问题需要考虑。Prolog 需要使用堆栈来储存递归的一些中间信息，当递归深入下去时，堆栈会越来越大。在本游戏中，由于参数较为复杂，堆栈是很容易溢出的。

幸运的是，Prolog 对于这种类型的递归有优化的方法。

尾递归

递归有两种类型。在真正的递归程序中，每一层必须使用下一层调用返回的信息。这意味着 Prolog 必须建立堆栈来储存每一层的信息。

这与重复操作是不同的，在通常的语言中，我们一般使用的是重复操作。重复操作只需要把信息传递下去就行了，而不需要保存每一次调用的信息。我们可以使

用递归来实现重复，这种递归就叫做尾递归。它的通常的形式是递归语句在最后，每一层的计算不需要使用下一层的返回信息，所以在这种情况下，好的 Prolog 解释器不需要使用堆栈。

计算阶乘就属于尾递归类型。首先我们使用通常的递归形式。注意从下一层返回的变量 FF 的值被使用到了上一层。

```
factorial_1(1,1).

factorial_1(N,F):-

N > 1,

NN is N - 1,

factorial_1(NN,FF),

F is N FF.



?- factorial_1(5,X).

X = 120
```

如果引入一个新的变量来储存前面调用的结果，我们就可以把 factorial/3 写成尾递归的形式。新的参数的初始值为 1。每次递归调用将计算第二个参数的值，当到达了边界条件，第三个参数就绑定为第二个参数。

```
factorial_2(1,F,F).

factorial_2(N,T,F):-

N > 1,

TT is N T,

NN is N - 1,

factorial_2(NN,TT,F).
```

```
?- factorial_2(5,1,X).

X = 120
```

它的结果和前面的相同，不过由于使用了尾递归，就不需要使用堆栈来储存中间的信息了。

把列表的元素顺序倒过来的谓词也可以使用尾递归来完成。

```
naive_reverse([],[]).

naive_reverse([H|T],Rev):-

naive_reverse(T,TR),

append(TR,[H],Rev).



?- naive_reverse([ants, mice, zebras], X).

X = [zebras, mice, ants]
```

这个谓词在逻辑上是完全正确的，不过它的运行效率非常低。所以我们把它叫做原始(naive)的递归。

当引入一个用来储存部分运算结果的新的参数后，我们就可以使用尾递归来重写这个谓词。

```
reverse([], Rev, Rev).

reverse([H|T], Temp, Rev) :-

reverse(T, [H|Temp], Rev).
```

```
?- reverse([ants, mice, zebras], [], X).

X = [zebras, mice, ants]
```

第 17 章-自然语言

prolog-使用逻辑编程语言的教程，第十七章 - 自然语言。主要包括自然语言理解，差异表，Definite Clasue Grammar(DCG)等。

Prolog 特别适合开发自然语言的应用系统。在这一章，我们将为寻找 Nani 游戏添加自然语言理解的部分。（由于 Prolog 谓词是使用的英文符号，所以这里的自然语言理解只能局限在英文中）

在着手于编制寻找 Nani 之前， 我们先来开发一个能够分析简单英语句子的模块。把这种方法掌握之后，编制寻找 Nani 的自然语言部分就不在话下了。

下面是两个简单的英语句子：

The dog ate the bone.

The big brown mouse chases a lazy cat.

我们可以使用下面的语法规则来描述这种句子。

sentence : (句子)

nounphrase, verbphrase.

nounphrase : （名词短语）

determiner, nounexpression.

nounphrase : （名词短语）

nounexpression.

nounexpression :

```
noun.

nounexpression :

adjective（形容词）, nounexpression.

verbphrase :（动词短语）

verb, nounphrase.

determiner :（限定词）

the | a.

noun :（名词）

dog | bone | mouse | cat.

verb :（动词）

ate | chases.

adjective :

big | brown | lazy.
```

稍微解释一下：第一条规则说明一个句子有一个名词短语和一个动词短语构成。最后的一个规则定义了单词 big、brown 和 lazy 是形容词，中间的“|”表示或者的意思。

首先，来判断某个句子是否是合法的句子。我们编写了 sentence/1 谓词，它可以判断它的参数是否是一个句子。

句子必须用 Prolog 的一种数据结构来表达，这里使用列表。例如，前面的两个句子的 Prolog 表达形式如下：

```
[the,dog,ate,the,bone]
```

[the,big,brown,mouse,chases,a,lazy,cat]

分析句子的方法有两种。第一种是选择并校样的方法（见后面的人工智能实例部分），使用这种方法，首先把句子的可能分解情况找出来，再来测试被分解的每一个部分是否合法。我们前面已经介绍过使用 append/3 谓词能够把列表分成两个部分。使用这种方法，顶层的规则可以是如下的形式：

```
sentence(L) :-
append(NP, VP, L),
nounphrase(NP),
verbphrase(VP).
```

append/3 谓词可以把列表 L 的所有可能的分解情况穷举出来，分解后的两个部分为 NP 和 VP，其后的两个目标则分别测试 NP 和 VP 是否是合法的，如果不是则会产生回溯，从而测试其他的分解情况。

谓词 nounphrase/1 和 verbphrase/1 的编写方法与 sentence/1 基本相同，它们调用其他的谓词来判断句子中的更小的部分是否合法，只到调用到定义单词的谓词，例如：

```
verb([ate]).
verb([chases]).

noun([mouse]).
noun([dog]).
```

差异表

前面的这种方法效率是非常低的，这是因为选择并校验的方法需要穷举所有的情况，更何况在每一层的目标之中都要进行这种测试。

更有效的方法就是跳过选择的步骤，而直接把整个列表传到下一级的谓词中，每个谓词把自己所要寻找的语法元素找出来，并返回剩下的列表。

为了能够达到这个目标，我们需要介绍一种新的数据结构：差异表。它由两个相关的表构成，第一个表称为全表，而第二个表称为余表。这两个表可以作为谓词的两个参数，不过我们通常使用‘-’连接这两个表，这样易于阅读。它的形式是 X-Y。

我们使用差异表改写了第一条语法规则。如果能够从列表 S 的头开始，提取出一个名词短语，其余部分 S1,并且能够从 S1 的头开始，提取出一个动词短语，并且其余部分为空表，那么列表 S 是一个句子。（这句话要细心理解，差异表所表示的表是全表和余表之间的差异。）

```
sentence(S) :-

nounphrase(S-S1),

verbphrase(S1-[]).
```

我们先跳过谓词 nounphrase/1 和 verbphrase/1 的编写，而来看看是如何定义真正的单词的。这些单词也必须书写成差异表的形式，这个很容易做到：如果列表的第一个元素是所需的单词，那么余表就是除去第一个单词的表。

```
noun([dog|X]-X).

noun([cat|X]-X).

noun([mouse|X]-X).
```

```
verb([ate|X]-X).
verb([chases|X]-X).

adjective([big|X]-X).
adjective([brown|X]-X).
adjective([lazy|X]-X).

determiner([the|X]-X).
determiner([a|X]-X).
```

下面是两个简单的测试，

```
?- noun([dog,ate,the,bone]-X).
%第一个单词 dog 是名词，于是成功，并且余表是后面的元素组成的表。
X = [ate,the,bone]

?- verb([dog,ate,the,bone]-X).
no
```

我们把剩下的一些语法规则写完：

```
nounphrase(NP-X):-
determiner(NP-S1),
```

```
nounexpression(S1-X).


nounphrase(NP-X):-

nounexpression(NP-X).


nounexpression(NE-X):-

noun(NE-X).


nounexpression(NE-X):-

adjective(NE-S1),

nounexpression(S1-X).


verbphrase(VP-X):-

verb(VP-S1),

nounphrase(S1-X).
```

注意谓词 nounexpression/1 的递归定义，这样就可以处理名词前面有任意多个形容词的情况。

我们来用几个句子测试一下：

```
?- sentence([the,lazy,mouse,ate,a,dog]).

yes
```

```
?- sentence([the,dog,ate]).
no

?- sentence([a,big,brown,cat,chases,a,lazy,brown,dog]).
yes

?- sentence([the,cat,jumps,the,mouse]).
no
```

下面是单步跟踪某个句子的情况：

询问是

```
?- sentence([dog,chases,cat]).

1-1 CALL sentence([dog,chases,cat])
2-1 CALL nounphrase([dog,chases,cat]-_0)
3-1 CALL determiner([dog,chases,cat]-_0)
3-1 FAIL determiner([dog,chases,cat]-_0)
2-1 REDO nounphrase([dog,chases,cat]-_0)
3-1 CALL nounexpression([dog,chases,cat]- _0)
4-1 CALL noun([dog,chases,cat]-_0)
4-1 EXIT noun([dog,chases,cat]-
[chases,cat])
```

注意，表示余表的变量的绑定操作是直到延伸至最底层时才进行的，每一层把它的余表和上一层的绑定。这样，当到达了词汇层时，绑定的值将通过嵌套的调用返回。

3-1 EXIT nounexpression([dog,chases,cat]-

[chases,cat])

2-1 EXIT nounphrase([dog,chases,cat]-

[chases,cat])

现在已经找出了名词短语，下面来测试余表是否为动词短语。

2-2 CALL verbphrase([chases,cat]-[])

3-1 CALL verb([chases,cat]-_4)

3-1 EXIT verb([chases,cat]-[cat])

很容易地就找出了动词，下面寻找最后的动词短语。

3-2 CALL nounphrase([cat]-[])

4-1 CALL determiner([cat]-[])

4-1 FAIL determiner([cat]-[])

3-2 REDO nounphrase([cat]-[])

4-1 CALL nounexpression([cat]-[])

5-1 CALL noun([cat]-[])

5-1 EXIT noun([cat]-[])

4-1 EXIT nounexpression([cat]-[])

3-2 EXIT nounphrase([cat]-[])

```
2-2 EXIT verbphrase([chases,cat]-[])

1-1 EXIT sentence([dog,chases,cat])

yes
```

寻找 nani

现在将使用这种分析句法结构的技术，来完成寻找 Nani。

我们首先假设已经完成以下的两个任务。第一，已经完成了把用户的输入转换成列表的工作。第二，我们可是使用列表的形式来表示命令，例如，goto(office)表示成为[goto,office]，而 look 表示成为[look]。

有了这两个假设，现在的任务就是把用户的自然语言转换成为程序能够理解的命令列表。例如，我们希望程序能够把[go,to,the,office]转换成为[goto,office]。

最高层的谓词叫做 command/2，它的形式如下：

```
command(OutputList, InputList).
```

最简单的命令就是只有一个动词的命令，例如 look、list_possessions 和 end。我们可以使用下面的子句来识别这种命令：

```
command([V], InList):- verb(V, InList-[]).
```

我们使用前面介绍过的方法来定义动词，不过这次将多加入一个参数，这个参数用来构造返回的标准命令列表。为了使这个程序看上去更有趣，我们让它能够识别命令多种表达形式。例如结束游戏可以输入：end、quit 和 good bye。

下面是几个简单的测试：

```
?- command(X,[look]).

X = [look]


?- command(X,[look,around]).

X = [look]


?- command(X,[inventory]).

X = [list_possessions]


?- command(X,[good,bye]).

X = [end]
```

下面的任务要复杂一些，我们将考虑动宾结构的命令。使用前面介绍过的知识，可以很容易地完成这个任务。不过此处，还希望除了语法以外还能够识别语义。

例如，goto 动词后面所跟随的物体必须是一个地方，而其他的谓词后面的宾语则是个物体。为了完成这个任务，我们引入了另一个参数。

下面是主子句，我们可以看出新的参数是如何工作的。

```
command([V,O], InList) :-
```

```
verb(Object_Type, V, InList-S1),

object(Object_Type, O, S1-[]).
```

还必须用事实来定义一些新的动词：

```
verb(place, goto, [go,to|X]-X).

verb(place, goto, [go|X]-X).

verb(place, goto, [move,to|X]-X).
```

我们甚至可以识别 goto 动词被隐含的情况，即如果玩家仅仅输入某个房间的名称，而没有前面的谓词。这种情况下列表及其余表相同。而 room/1 谓词则用来检测列表的元素是否为一个房间，除了房间的名字是两个单词的情况。

下面这条规则的意思是：如果我们从列表的头开始寻找某个动词，而列表的头确是一个房间的名称，那么就认为找到了动词 goto，并且返回完成的列表，好让后面的操作找到 goto 动词的宾语。

```
verb(place, goto, [X|Y]-[X|Y]):- room(X).

verb(place, goto, [dining,room|Y]-[dining,room|Y]).
```

下面是关于物品的谓词：

```
verb(thing, take, [take|X]-X).

verb(thing, drop, [drop|X]-X).

verb(thing, drop, [put|X]-X).
```

```
verb(thing, turn_on, [turn,on|X]-X).
```

有时候,物品前面可能有限定词,下面的两个子句考虑的有无限定词的两种情况:

```
object(Type, N, S1-S3) :-
det(S1-S2),
noun(Type, N, S2-S3).
object(Type, N, S1-S2) :-
noun(Type, N, S1-S2).
```

由于我们处理句子时只需要去掉限定词，所以就不需要额外的参数。

```
det([the|X]- X).
det([a|X]-X).
det([an|X]-X).
```

定义名词的方法与动词相似，不过大部分可以使用原来的定义方法，而只有那些两个单词以上的名词才需要特殊的定义方法。位置名词使用 room 谓词定义。

```
noun(place, R, [R|X]-X):- room(R).
noun(place, 'dining room', [dining,room|X]-X).
```

location 谓词和 have 谓词所定义的东西是物品，这里我们又必须把两个单词的物品单独定义。

```
noun(thing, T, [T|X]-X):- location(T,_).

noun(thing, T, [T|X]-X):- have(T).

noun(thing, 'washing machine', [washing,machine|X]-X).
```

我们可以把对游戏当前状态的识别也做到语法中去。例如，我们想做一个可以开关灯的命令，这个命令是 turn_on(light)，和 turn_on(flashlight)相对应。如果玩家输入 turn on the light，我们必须决定这个 light 是指房间里的灯还是 flashlight。

在这个游戏中，房间的灯是永远也打不开的，因为玩家所扮演的角色是一个 3 岁的小孩，不过她可以打开手电筒。下面的程序把 turn on the light 翻译成 turn on light 或者 turn on flashlight，这样就能让后面的程序来进行判断了。

```
noun(thing, flashlight, [light|X], X):- have(flashlight).

noun(thing, light, [light|X], X).
```

下面来全面的测试一下：

```
?- command(X,[go,to,the,office]).

X = [goto, office]


?- command(X,[go,dining,room]).


X = [goto, 'dining room']
```

```
?- command(X,[kitchen]).

X = [goto, kitchen]




?- command(X,[take,the,apple]).

X = [take, apple]


?- command(X,[turn,on,the,light]).

X = [turn_on, light]


?- asserta(have(flashlight)), command(X,[turn,on,the,light]).

X = [turn_on, flashlight]
```

下面的几个句子不合法：

```
?- command(X,[go,to,the,desk]).

no


?- command(X,[go,attic]).

no


?- command(X,[drop,an,office]).

no
```

Definite Clasue Grammar(DCG)

在 Prolog 中经常用到差异表，因此许多 Prolog 版本都对差异表有很好的支持，这样就可以隐去差异表的一些繁琐复杂之处。这种语法称为 Definite Clasue Grammer(DCG)，它看上去和一般的 Prolog 子句非常相似，只不过把连接符:-替换成为-->，这种表达形式由 Prolog 翻译成为普通的差异表形式。

使用 DCG，原来的句子谓词将写为：

```
sentence --> nounphrase, verbphrase.
```

这个句子将被翻译成一般的使用差异表的 Prolog 子句，但是这里不再用“-”隔开，而是变成了两个参数，上面的这个句子与下面的 Prolog 子句等价。

```
sentence(S1, S2):-
nounphrase(S1, S3),
verbphrase(S3, S2).
```

因此，既是使用 DCG 形式定义 sentence 谓词，我们在调用时仍然需要两个参数。

```
?- sentence([dog,chases,cat], []).
```

用 DCG 来表示词汇只需要使用一个列表：

```
noun --> [dog].
verb --> [chases].
```

这两个句子被翻译成：

```
noun([dog|X], X).

verb([chases|X], X).
```

就象在本游戏中所需要的那样，有时需要额外的参数来返回语法信息。这个参数只需要简单地加入就行了，而句中纯 Prolog 则使用{}括起来，这样 DCG 分析器就不会翻译它。游戏中的复杂的规则将写成如下的形式：

```
command([V,O]) -->

verb(Object_Type, V),

object(Object_Type, O).


verb(place, goto) --> [go, to].

verb(thing, take) --> [take].


object(Type, N) --> det, noun(Type, N).

object(Type, N) --> noun(Type, N).


det --> [the].

det --> [a].


noun(place,X) --> [X], {room(X)}.
```

```
noun(place,'dining room') --> [dining, room].

noun(thing,X) --> [X], {location(X,_)}.
```

由于 DCG 自动的取走第一个参数，如果只输房间名称，前面的子句就不能起作用，所以我们还要加上一条：

```
command([goto, Place]) --> noun(place, Place).
```

读入句子

让我们来最后完工吧。最后的工作是把用户的输入变成一张表。下面的程序很够完成这个任务：

```
% read a line of words from the user


read_list(L) :-
write('> '),
read_line(CL),
wordlist(L,CL,[]), !.


read_line(L) :-
get0(C),
buildlist(C,L).


buildlist(13,[]) :- !.
buildlist(C,[C|X]) :-
```

```
get0(C2),

buildlist(C2,X).



wordlist([X|Y]) --> word(X), whitespace, wordlist(Y).

wordlist([X]) --> whitespace, wordlist(X).

wordlist([X]) --> word(X).

wordlist([X]) --> word(X), whitespace.



word(W) --> charlist(X), {name(W,X)}.



charlist([X|Y]) --> chr(X), charlist(Y).

charlist([X]) --> chr(X).



chr(X) --> [X],{X>=48}.



whitespace --> whsp, whitespace.

whitespace --> whsp.



whsp --> [X], {X48}.
```

它包括两个部分：首先使用内部谓词 get0/1 读入单个的 ASCII 字符， ASCII 13 代表句子结束。第二部分使用 DCG 分析字符列表，从而把它转化为单词列表，这里使用了另一个内部谓词 name/2，它把有 ASCII 字符组成的列表转化为原子。

另外一部分是把形如[goto,office]的命令，转化为 goto(office)，我们使用称为 univ 的内部谓词完成这个工作，使用"=.."表示。它的作用如下，把一个谓词转化为了一个列表，或者反过来。

```
  ?- pred(arg1,arg2) =.. X.

X = [pred, arg1, arg2]



?- pred =.. X.

X = [pred]



?- X =.. [pred,arg1,arg1].

X = pred(arg1, arg2)



?- X =.. [pred].

X = pred
```

最后我们使用前面的两个部分做成一个命令循环：

```
get_command(C) :-

read_list(L),

command(CL,L),

C =.. CL, !.



get_command(_) :-

write('I don''t understand'), nl, fail.
```

到此为止，我们的第章就全部结束了，但是你的工作没有结束，如果想很好地掌握这门语言，还有很漫长的路要走。

非常感谢您阅读完这个教程，接下来就请进入实战部分吧。

第 18 章 C 语言调用 Prolog Amzi 逻辑服务器

本文主要介绍如何使用 C 语言调用 Prolog 的 Amzi 逻辑服务器，给出了 C 语言的源代码。

Prolog 的 Amzi 的逻辑服务器提供了 C 和 C++的调用界面。这里我们着重介绍 C 语言的调用界面,本章使用 C++ builder 作为 C 语言编译器。在正式编写调用 Amzi 逻辑服务器的程序之前，你需要做如下的工作。

找到 bin 目录下面的 amzi.dll 文件，并且复制到你的 windows 目录下。

找到 include 目录下的 amzi.h 文件，并且复制到 c++ builder 所在文件夹的 include 子目录下。

找到 lib 目录下的 amzib.lib 文件，复制到你的程序文件所在的目录。

作为测试，把 samples/c/hello 下的 hello.xpl 也复制到程序文件所在的目录。

在你的程序中加入#include

选择 c++ builder 菜单中的 project/add to project，把文件 amzib.lib 添加入你的工程中。

下面开始编写测试 Amzi 逻辑服务器的程序代码部分:

在 form 中添加一个 Tlabel，名字为 Label1，

然后加入如下程序段落：

```
1.  void  __fastcall  TForm1::FormCreate(TObject  Sender)
2.  {
3.  char  buf[120];
4.  ENGid  CurEng;
5.  TERM  t;  /*  a  Prolog  term  */
```

```
6.  RC rc; /* LSAPI return code */
7.  TF tf; /* LSAPI true/false/err return codes */
8.  /* Initialize and load the compiled Prolog program */

9.  lsInit(&CurEng, "hello");
10. lsLoad(CurEng, "hello");
11. /* Build a query term and call it */
12. tf = lsCallStr(CurEng, &t, "hello($C Programmer$, X)");

13. /* If the query succeeded print up the results */

14. lsGetArg(CurEng, t, 2, cSTR, buf);
15. Label1->Caption=buf;
16. lsClose(CurEng);
17. }
```

运行程序之后，会看到 Label1 的文字变为 Greetings C Programmer, from Amzi! Prolog. 这样就表示你的所有设置都已经正确了。

我们来分析一下上面所做的事情。

amzi.dll 是个动态连接库，其中包含了逻辑服务器提供的所有函数功能，所以一定要把它放到你的程序能够找到的地方(例如 windows 目录下，或者你的程序目录下)，你的程序才能够运行。

Amzi.h 中声明了 amzi.dll 中的 API 函数，以及一些特殊的数据形式。所以你的程序必须 include 这个文件。

Amzib.lib 是编译连接时的库文件，所以你必须把它添加到你的工程中间去。

有了以上的操作就可以调用逻辑服务器了。

在使用逻辑服务器之前，必须对它初始化。初始化服务器的函数是：lsInit，它的定义形式如下：

```
RC lsInit(ENGid cureng, STRptr ininame);
```

RC 数据类型的定义是：

```
typedef int RC;
```

也就是说 lsInit 返回整数类型的数据，0 代表初始化成功，其他的值都是错误代码，代表不同的错误信息。

NGid 其实是一个指针，当使用 lsInit 初始化服务器以后，就分配一块内存区域，而 ENGid 就是指向这区域的指针。也就是说你可以初始化几个逻辑服务器，分别使用不同的指针指向它们，这些逻辑服务器之间是相互独立的，就像你能够同时打开多个文件一样。第二个参数是初始化文件的文件名。初始化文件的扩展名为 ini，其中定义了服务器说占用的系统资源的大小。若找不到这个文件，系统将使用缺省参数初始化服务器。

所以我们初始化服务器的程序片段如下：

```
ENGid CurEng;

lsInit(&CurEng, "hello");
```

这样就初始化了一个逻辑服务器，并且用 CurEng 代表它，所以以后想对此服务器进行任何操作，都必须使用 CurEng 指明是使用这个服务器。

注意，初始化逻辑服务器之后不能够直接使用它，必须调入某个已经编译连接过的 prolog 文件，扩展名为 xpl。使用 lsLoad 函数调入 xpl 文件，调用形式如下：

```
lsLoad(CurEng, "hello");
```

这句话表示把文件 hello.xpl 调入到 CurEng 服务器中。

然后就可以对你调入的 prolog 文件进行询问了。如何把询问命令传递给服务器呢，这里使用的是 lsCallStr，它的调用形式如下：

```
TF tf;

TERM t;

tf = lsCallStr(CurEng, &t, "hello($C Programmer$, X)")
```

返回 TF 数据类型的数据，TF 就是代表 true，false，即真假，每个 prolog 子句都只能够返回这两个值中间的一个，返回值为 1 表示查询的子句成功，0 表示失败。

第一个参数是服务器指针。

第三个参数是字符串，即需要查询的内容。

下面详细介绍一下第二个参数的含义。

为了能够有效的使用逻辑服务器，必须能够在 C 语言和 Prolog 之间传递信息，上面介绍的第三个参数可以把字符串传递给逻辑服务器，从而让服务器进行某项工作，那么如何把服务器的结果返回到 C 语言的程序中呢，这就是第二个参数的功能。第二个参数的数据类型是 TERM。

在 Prolog 中，所有的数据形式都使用同样一种机构表达，这种机构称为 term。下面的数据类型在 Prolog 中都是 term。

```
X
foo(a,b)
hello
$this is a string$
'a long atom name'
```

```
[a,list,of,stuff,34,$foo$,foo(a,b),Z,Y]
42
3.7
13 + 8.9 / 16
even_rules(X) :- are(Y,Z), terms
```

在 prolog 中 term 实质上是一个指针，它能够指向众多类型的数据，从最简单的整数型数据，到非常复杂的列表都可以指向。这个指针就是在两种语言中传递信息的桥梁。我们的 hello.pro 程序如下：

```
hello(Caller, Greeting) :-

strcat($Greetings $, Caller, S1),

strcat(S1, $, from Amzi! Prolog.$, Greeting).
```

如果调用 hello($C Programmer$, X)，那么 X 将绑定为

$Greetings C Programmer, from Amzi! Prolog.$。

从而整个子句变成了

```
hello($C Programmer$ , $Greetings C Programmer, from Amzi! Prolog.$).
```

而这个子句实际上是一种数据结构，储存在 CurEng 所指向的服务器的内存空间中，并且使用某个 term 指针指向它。而这个指针就返回到了 lsCallStr 的第二个参数中。

```
tf = lsCallStr(CurEng, &t, "hello($C Programmer$, X)")
```

注意，term 所指向的是一个 Prolog 的数据结构，所以在 C 语言中还不能够直接使用它，还需要特别的函数把这个 term 指针所指向的 Prolog 数据结构转换为 C 语言中的数据。常用的转换函数有如下几个：

```
RC lsTermToStr(ENGid cureng, TERM term, STRptr str, int len);
```

这个函数把 Prolog 的 term 所指向的数据转换为 C 语言中的字符串。

第一个参数是服务器指针，第二个参数是这个服务器中的某个 term 的指针，第三个参数是字符串指针，第四个参数是字符串的长度（为了防止溢出）。

所以下面的程序就可以把结果返回到 C 语言中：

```
char buff[100];

tf = lsCallStr(CurEng, &t, "hello($C Programmer$, X)");

lsTermToStr(CurEng,t,buff,100);
```

这样 buff 的值就为：

“hello($C Programmer$, $Greetings C Programmer, from Amzi! Prolog.$).”

有时候并不不要取得整个 term 的数据的值，在上例中，也许我们只关心最后那个变量绑定成什么值了，也就是 hello 的第二个参数值。函数 lsGetArg 就可以完成这个任务。

```
lsGetArg(CurEng, t, 2, cSTR, buf);
```

第一个参数是服务器指针；第二个参数是 term 指针；第三个参数是需要取得的参数序号，在上例中为 2，表示我们需要取得 hello 的第二个参数；cSTR 告诉 lsGetArg 转换的目标数据类型，这里是 cSTR，表示把 hello 的第二个参数转换为字符串类型，最后的 buf 是字符串指针。

当运行上面的这个函数之后，buf 的值为：

“Greetings C Programmer, from Amzi! Prolog.”

最后需要关闭逻辑服务器，

lsClose(CurEng);

好了，到此为止就介绍了最基本的调用逻辑服务器的方法。

下一章介绍如何使用 C 语言扩展 prolog 的功能，也就是如何让 prolog 调用 C 函数。

第 19 章 Prolog 调用 C 语言 - 以扩展谓词为例

prolog 在逻辑推理方面功能强大,而其他的方面则有些捉襟见肘了。本章介绍如何使用 C 语言扩展 prolog 的功能。

为了让 prolog 调用 C 的函数，就必须使用扩展谓词，这些扩展谓词用起来就和其他的内部谓词一样，只不过他们的代码是用户使用别的语言编写的。

为了使用扩展谓词，你必须做下面两件事情:

定义执行谓词功能的 C 语言函数;

在初始化的时候通知逻辑服务器扩展谓词的谓词名，以及它所对应的 C 语言函数。

下面我们来分别介绍这两个部分。

定义扩展谓词

在我们的例子中，编写了三个有关数组的扩展谓词：

make_array/2 生成一个数组，第一个参数是变量，调用后绑定为此数组，第二个参数为数组大小。

array_elem/3 数组元素的读出和写入，第一个参数为变量，代表要操作的数组，第二个参数为要读出或写入的元素序号,第三个参数若为变量,则绑定为元素值，若为数值，则把此数值写入数组。

delete_array/1 从内存中删除数组。参数为绑定为某个数组的变量。

扩展谓词对应的 C 语言函数只有一个参数,这个参数就是调用此扩展谓词的逻辑服务器的指针。

make_array/2 所对应的 C 语言函数如下：

```
1.  TF  EXPFUNC  pMakeArray(ENGid  eid)
2.  {
3.          long  iArray;
4.          int  iSize;
5.          TERM  t;
6.          lsGetParm(eid,  2,  cINT,  &iSize);  /*  get  size  o
    f  array  */
```

```
7.          iArray = new long[iSize];
8.          lsMakeAddr(eid, &t, iArray); /* send back addre
   ss of array */
9.          lsUnifyParm(eid, 1, cTERM, &t);
10.         return TRUE;
11. }
```

函数的定义形式如下：

TF EXPFUNC functionname(ENGid eid);

那么如何获得从 prolog 传递过来的参数呢？这里需要使用函数

RC lsGetParm(ENGid cureng, int iparm, cTYPE ctype, VOIDptr valp);

第一个参数是服务器指针，第二个参数是要获得的参数序号，第三个参数是需要把 prolog 传递过来的参数转换成 C 中的何种数据类型，最后一个参数用来保存转换以后的数据。例如

lsGetParm(eid, 2, cINT, &iSize);

的意思就是把第二个参数转化为整数，并且保存到 iSize 变量中。

接下来使用 C 语言的 new 分配一段内存空间，

iArray = new long[iSize];

然后把这个数组指针转换为 Prolog 的 term：

lsMakeAddr(eid, &t, iArray);

(注意：由于 iArray 的数值实际上是一个地址，所以使用 lsMakeAddr 转换，还有一系列的 lsMake 函数,能够把不同的 C 的数据类型转换为 Prolog 中的 Term。)

最后让 prolog 传递过来的第一个参数绑定为这个 Term：

lsUnifyParm(eid, 1, cTERM, &t);

lsUnifyParm 是从 C 返回信息到 Prolog 中的函数。上面的语句的含义是：

把第一个参数绑定为 t 的值，也就是数组指针的值。通过上面的操作就把在 C 中分配的一段内存空间的地址传递给了 Prolog。

定义完扩展谓词之后，需要通知 Prolog 我们已经定义好了扩展谓词。

在你的 c 语言的程序中，加入如下的全局变量

```
PRED_INIT arrayPreds[] =
{
{"make_array", 2, pMakeArray},
{"array_elem", 3, pArrayElem},
{"delete_array", 1, pDeleteArray},
{NULL, 0, NULL}
};
```

{"make_array", 2, pMakeArray},表示把 C 语言的函数 pMakeArray 映射成为 prolog 的谓词 make_array，并且这个谓词有两个参数。

最后在你初始化服务器的时候加入下面的语句：

```
lsInit(&cureng, "xarray");
InitPreds(cureng, arrayPreds);
lsLoad(cureng, "xarray");
```

上面的第二句的函数 InitPreds，就是告诉服务器 cureng 我们的扩展谓词的定义在字符串 arrayPreds 中。注意这个函数必须在 lsLoad 调用之前调用。

这样我们就可以在 prolog 中调用 C 语言的程序了。

本例中的 prolog 程序如下：

```
test(Z) :-
make_array(A, 5), % create array of 5 elements
array_elem(A, 3, 9), % set third element to 9
array_elem(A, 4, 16), % set fourth element to 16
```

```
array_elem(A, 3, X), % retrieve third element
array_elem(A, 4, Y), % retrieve fourth element
Z is X+Y,
delete_array(A).
```

调用这段 prolog 程序的 C 语言程序如下：

```
1.  lsInit(&cureng,  "xarray");
2.  lsInitPreds(cureng,  arrayPreds);
3.  lsLoad(cureng,  "xarray");
4.  tf  =  lsCallStr(cureng,&t,"test(X)");
5.  lsGetArg(cureng,t,1,cINT,&x);
6.  Label1->Caption=x;
7.  lsClose(cureng);
```

整个的调用情况如下图所示：

当然也可以把编写的扩展谓词程序编译成 DLL 文件，然后重命名为 XPL 文件，在 amzi prolog listener 中直接使用这些扩展谓词，这一部分内容暂时不做介绍，请参照 amzi 所附带的 help，自己研究。这种调用情况如下图所示。